

File Systems

↻ **Measuring the Elusive Working Set in Storage**

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield

↻ **Container Farms and Storage**

Mark Lamourine

↻ **Finding Faults with Error Handlers to Avoid Catastrophic Failures**

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm

↻ **Nail, A Secure Parser/Generator**

Julian Bangert and Nikolai Zeldovich

Columns

Practical Perl Tools: RESTful Clients Using Perl

David N. Blank-Edelman

Python Tricks for Checking Function Argument Types

David Beazley

iVoyeur: Using Graphite and Statsd to Preserve Wide Data

Dave Josephsen

For Good Measure: Cyber-job Security and Automation

Dan Geer

/dev/random: Smarter-Than-You Storage—the Future of Storage

Robert G. Ferrell

Conference Reports

11th USENIX Symposium on Operating Systems Design and Implementation

2014 Conference on Timely Results in Operating Systems

10th Workshop on Hot Topics in System Dependability

SRECon15

March 16–17, 2015, Santa Clara, CA, USA
www.usenix.org/srecon15

NSDI '15: 12th USENIX Symposium on Networked Systems Design and Implementation

May 4–6, 2015, Oakland, CA, USA
www.usenix.org/nsdi15

SREcon15 Europe

May 14–15, 2015, Dublin, Ireland

HotOS XV: 15th Workshop on Hot Topics in Operating Systems

May 18–20, 2015, Kartause Ittingen, Switzerland
www.usenix.org/hotos15

USENIX ATC '15: 2015 USENIX Annual Technical Conference

July 8–10, 2015, Santa Clara, CA, USA
www.usenix.org/atc15

Co-located with USENIX ATC '15 and taking place July 6–7, 2015:

HotCloud '15: 7th USENIX Workshop on Hot Topics in Cloud Computing

Submissions due March 10, 2015
www.usenix.org/hotcloud15

HotStorage '15: 7th USENIX Workshop on Hot Topics in Storage and File Systems

Submissions due March 17, 2015
www.usenix.org/hotstorage15

USENIX Security '15: 24th USENIX Security Symposium

August 12–14, 2015, Washington, D.C., USA
www.usenix.org/usenixsecurity15

Co-located with USENIX Security '15:

WOOT '15: 9th USENIX Workshop on Offensive Technologies

August 10–11, 2015
www.usenix.org/woot15

CSET '15: 8th Workshop on Cyber Security Experimentation and Test

August 10, 2015
Submissions due April 23, 2015
www.usenix.org/cset15

FOCI '15: 5th USENIX Workshop on Free and Open Communications on the Internet

August 10, 2015
Submissions due May 12, 2015
www.usenix.org/foci15

HealthTech '15: 2015 USENIX Summit on Health Information Technologies

Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 10, 2015
www.usenix.org/healthtech15

JETS '15: 2015 USENIX Journal of Election Technology and Systems Workshop

(Formerly EVT/WOTE)
August 11, 2015
www.jets-journal.org

HotSec '15: 2015 USENIX Summit on Hot Topics in Security

August 11, 2015
www.usenix.org/hotsec15

2015 USENIX Summit on Gaming, Games, and Gamification in Security Education

August 11, 2015
Submissions due May 5, 2015
www.usenix.org/3gse15

LISA15

November 8–13, 2015, Washington, D.C., USA
Submissions due April 17, 2015
www.usenix.org/lisa15

Do you know about the USENIX Open Access Policy?

USENIX is the first computing association to offer free and open access to all of our conferences proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your membership fees play a major role in making this endeavor successful.

Please help us support open access. Renew your USENIX membership and ask your colleagues to join or renew today!

www.usenix.org/membership

Stay Connected...



twitter.com/usenix



www.usenix.org/youtube



www.usenix.org/gplus



www.usenix.org/facebook



www.usenix.org/linkedin



www.usenix.org/blog

;login:

FEBRUARY 2015 VOL. 40, NO. 1

EDITORIAL

- 2 Musings *Rik Farrow*

FILE SYSTEMS AND STORAGE

- 6 Counter Stacks and the Elusive Working Set
*Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey,
and Andrew Warfield*
- 10 Storage Options for Software Containers *Mark Lamourine*
- 15 Interview with Steve Swanson *Rik Farrow*

PROGRAMMING

- 18 Simple Testing Can Prevent Most Critical Failures: An Analysis
of Production Failures in Distributed Data-Intensive Systems
*Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues,
Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm*
- 24 Nail: A Practical Tool for Parsing and Generating Data Formats
Julian Bangert and Nikolai Zeldovich

SYSADMIN

- 32 Capacity Planning *David Hixson and Kavita Guliani*
- 39 /var/log/manager: Daily Perspectives for the Sysadmin
and the Manager *Andrew Seely*

COLUMNS

- 42 Practical Perl Tools: Give it a REST *David N. Blank-Edelman*
- 47 Thinking about Type Checking *David Beazley*
- 52 iVoyeur: Spreading *Dave Josephsen*
- 56 For Good Measure: Cyberjobsecurity *Dan Geer*
- 58 /dev/random: Smarter-Than-You Storage *Robert G. Ferrell*

BOOKS

- 60 Book Reviews *Mark Lamourine and Rik Farrow*

CONFERENCE REPORTS

- 62 11th USENIX Symposium on Operating Systems
Design and Implementation
- 86 2014 Conference on Timely Results in Operating Systems
- 93 10th Workshop on Hot Topics in System Dependability



EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street, Suite 215
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

\$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for non-members are \$90 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

©2015 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.



Rik is the editor of *login*:
rik@usenix.org

Back in the late nineties, I found myself sharing a Silicon Valley hotel Jacuzzi with a sales rep from a big hard-disk company. The sales rep was telling me that they soon expected to be selling 10-gigabyte hard drives to consumers, and I was astonished. Why in the world would people need such large drives in their desktop systems?

Now we can buy six-terabyte consumer drives—600 times as much capacity as I found unfathomable around 1999—for under \$300. And similar technology, in the smaller 2.5-inch form factor, fills server racks in many datacenters. Back when I thought that 10 gigabytes was a ridiculous amount, I wondered how mere mortals would manage all that storage. Well, turns out that I needn't have worried. Not only do most home users (and many businesses) not manage their storage, they can now expand out into apparently unlimited clouds of storage as well.

Life in the Clouds

To be honest, the problems with the storage surfeit is not just a cloud problem. During the presentation of one of my favorite storage papers [1], Dutch Meyer pointed out that many files found on Microsoft employees' desktops were "Write once, read never." Of the files that were modified, most were changed within one month of creation, then left alone. Forever. For storage vendors, this is a wonderful situation, as people will use ever larger amounts of storage since the supply appears endless. And now we can just store data "in the cloud," and if we forget about it, it is someone else's problem.

That's pretty nice compared to past methods for managing files that you or your organization had stored in case you might need the data again, some day. You've contracted with companies who keep redundant copies of your data, along with vague guarantees about how safe that is, but that's still better than the old days.

In the "old days," we managed our data by accident. Here's how that worked. We would store our data, carefully backing up what we considered important, until the hard drive (or RAID array or storage server) catastrophically failed. Then we would get out our backups, and see how well they worked when we tried to recover from the catastrophic failure.

Inevitably, much would be lost. But hey, that was storage management—the important data was either restored from backups, recreated from scratch, or the business or research project just failed. For home users, they'd just start over again with a brand new, mostly empty, and larger hard drive. See, storage management by accident.

We do have very serious uses for data, and I am purposely exaggerating. But there is more than a grain of truth in the problem of storage management, one that I believe still exists today.

The Lineup

We start out on the theme of storage with an article about measuring the size of the working set. The working set represents your hot data, the data you want to have ready for processing, within a relatively short interval. For programs working with big data, calculating the working set has been a real problem, as just collecting the block access data generates both a lot of data while adding a huge workload to the system under measurement.

Wires et al. describe a method for sampling block accesses and collecting enough information about those accesses to accurately measure the working set. I was impressed by this work and felt it was worth sharing with a larger audience than just those who read OSDI papers. I also have a hunch that their techniques will become ever more important as our storage requirements continue to grow.

Mark Lamourine offered to explain just how containers will complicate storage. When I read James Bottomley's "Containers" [2] article, I marveled at how we could now share resources safely without resorting to heavyweight VMs. I didn't understand the issues with long-term storage when containers are spawned in a farm of managed hosts, much like we fire up VMs in the cloud today.

I interviewed Steve Swanson about the past and future of non-volatile memory. NVM has gone from being almost unused to commonplace, mostly because of the work that has been done with flash. Steve explains how he became involved with flash, the problems vendors have needed to solve to make flash reliable, as well as directions for future research.

Heading off in a different direction, I was intrigued by Ding Yuan et al.'s work analyzing catastrophic failures of cluster software systems, like HBase and HDFS. Honestly, it was what they found that was amazing: that empty, over catching, or non-existent error handlers lead to most of the crashes in popular cluster software. Ding and his co-authors also produced (and shared) a tool, Aspirator, for finding bugs in error-handling.

Julian Bangert and Nicolai Zeldovich write about their tool, Nail, designed to build secure parsers. Many exploitable vulnerabilities, such as Heartbleed and bugs in the signature checking software in Android and iOS, involve failures in parsing. Nail solves these issues through being a tool for generating parsers, creating the data structures used while working with parsed data, as well as providing a method for correctly regenerating the processed data. Where almost all programs parse data, only Nail uses the same configuration for parsing, storing, and generating stored data.

Dave Hixson and Kavita Guliani continue the series of articles written about the practice of Google Site Reliability Engineers (SREs). Dave and Kavita have written about capacity planning, providing a very thorough approach that obviously comes from Hixson's hard-earned experience.

Andy Seely contrasts the worlds of the sysadmin and the technical manager. As he's worked in both positions, Andy does a great job of comparing the two viewpoints through illustrative stories.

David Blank-Edelman takes another look at REST, defining it and demonstrating several ways of making RESTful requests, including parsing the results, all with his trademark humorous style.

Dave Beazley takes a look at dynamic type handling. Python's flexibility can work against you because Python's ability to automatically convert types can cause unexpected behavior when calling functions. Dave has examples, including several ways to test that the expected types have been passed to functions.

Dave Josephsen continues on his theme of fat data. Dave doesn't want you to lose the richness of the data you are collecting through inappropriate ways of storing that data. In this column, Dave shows how to use a combination of Nagios, Graphite, StatsD, and Graphios to collect and save fat data.

Dan Geer takes on automation. While the future of computing (and manufacturing, farming, stock trading, and everything except service) appears to be automation, Dan questions the appropriateness of automation for dealing with crafty, and often state-sponsored, adversaries.

Robert Ferrell worked with the theme of storage and tells us what he considers will be the future of smart storage.

We have several book reviews by Mark Lamourine and myself. We also have summaries of OSDI '14 and two of the associated workshops.

Getting back to the topic of storage management, I believe that perhaps I've uncovered a region of computer science that is worthy of some serious research. Back when disk drive capacities were minuscule, another technique for storage management consisted of using a **find** command to list all files not accessed within the last chunk of time, say, six months. A bit of shell scripting later, users would receive a list of these "aged" files by email and need to request that they not be deleted. After these email warnings were ignored, the system administrator could then delete the files, then wait for the few frenzied requests for restoration of the missing files. Ah, those were the days!

I like to imagine that cloud providers actually do storage management at scale. I know that Facebook does, because they had a paper [3] about how they handle warm BLOBs (Binary Large Objects) by reducing the effective duplication factor. Note that Facebook is not throwing away rarely watched cat videos, just saving fewer copies of them. But how do you think Amazon, Google, and Rackspace handle their customers' warm, or cold, storage? They bill their customers for them.

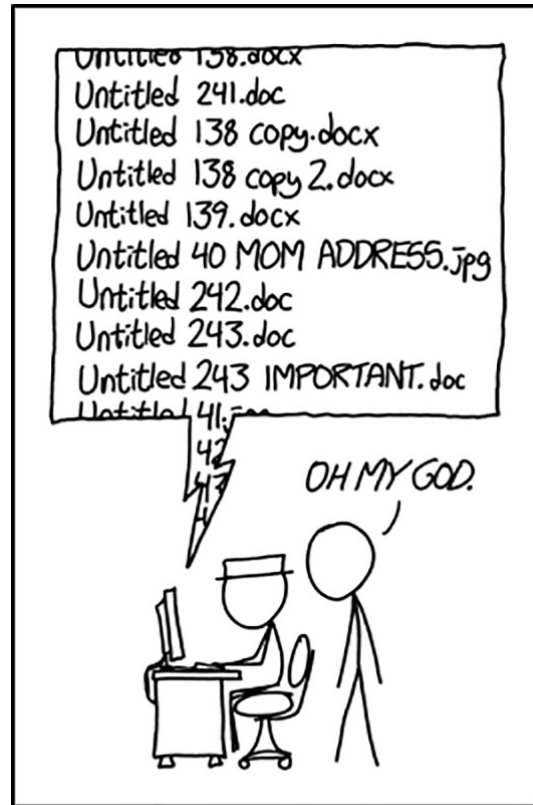
References

[1] D. Meyer, and W. Bolosky, "A Study of Practical Deduplication," FAST '11: https://www.usenix.org/legacy/event/fast11/tech/full_papers/Meyer.pdf.

[2] J. Bottomley and P. Emelyanov, "Containers," *login*, vol. 39, no. 5 (October 2014): <https://www.usenix.org/publications/login/oct14/bottomley>.

[3] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, Sanjeev Kumar, "Facebook's Warm BLOB Storage System": OSDI '14: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar>.


XKCD



PROTIP: NEVER LOOK IN SOMEONE ELSE'S DOCUMENTS FOLDER.








Announcing the USENIX Store!



Welcome to the USENIX Store!

Purchase USENIX-branded apparel and gear, copies of *;login:* issues and books from our Short Topics in System Administration series, and Video Box Sets from our USENIX conferences.

 <p>Apparel</p>	 <p>Gear</p>	
 <p><i>;login:</i> issues</p>	 <p>Short Topics Books</p>	 <p>Video Box Set USBs</p>

Want to buy a subscription to *;login:*, the latest short topics book, a USENIX or conference shirt, or the box set from last year's workshop? Now you can, via the brand new USENIX Store!

Head over to www.usenix.org/store and check out the collection of t-shirts, video box sets, *;login:* magazines, short topics books, and other USENIX and LISA gear. USENIX and LISA SIG members save, so make sure your membership is up to date.

www.usenix.org/store

Counter Stacks and the Elusive Working Set

JAKE WIRES, STEPHEN INGRAM, ZACHARY DRUDI, NICHOLAS J. A. HARVEY,
AND ANDREW WARFIELD



Jake Wires is a principal software engineer at Coho Data and a doctoral candidate at the University of British Columbia. He is broadly interested in the design of storage systems and scalable data processing. jake@cohodata.com



Stephen Ingram is a software engineer at Coho Data. He received his PhD from the University of British Columbia in 2013, his MSc from UBC in 2008, and his BSc Honors degree in computer science from Georgia Tech in 2004. His research interests are information visualization and dimensionality reduction. stephen@cohodata.com



Zachary Drudi is a software engineer at Coho Data. He completed his MSc in computer science at the University of British Columbia. Zach is interested in placing streaming algorithms in containers. zach@cohodata.com



Nick Harvey is a consultant at Coho Data and an assistant professor at the University of British Columbia. His main research area is algorithm design. He completed his PhD in computer science at MIT in 2008. nick@cohodata.com



Andrew Warfield is co-founder and CTO of Coho Data and an associate professor at the University of British Columbia. He is broadly interested in software systems. andy@cohodata.com

Counter stacks are a compact and effective data structure for summarizing access patterns in memory and storage workloads. They are a stream abstraction that efficiently characterizes the *uniqueness of an access stream over time*, and are sufficiently low overhead as to allow both new approaches to online decision-making (such as replacement or prefetching policies) and new applications of lightweight trace transmission and archiving.

A fascinating shift is currently taking place in the composition of datacenter memory hierarchies. The advent of new, nonvolatile memories is resulting in larger tiers of fast random-access storage that are much closer to the performance characteristics of processor caches and RAM than they are to traditional bulk-capacity storage on spinning disks. There are two very important consequences to this trend:

The I/O gap is narrowing. Historically, systems designers have been faced with a vast and progressively widening gulf between the access latencies of RAM (~10 ns) and that of spinning disks (~10 ms). Storage-class memories (SCMs) are changing this by providing large, nonvolatile memories that are more similar to RAM than disk from a performance perspective.

Memory hierarchies are stratifying. SCMs are being built using different types of media, including different forms of NAND flash and also newer technologies such as Memristor and PCM. These memories also attach to the host over different interfaces, including traditional disk (SAS/SATA), PCIe/NVMe, and even the DIMM bus on which RAM itself is connected. These offerings have a diverse range of available capacities and performance levels, and a correlated range of prices. As a result, the memory hierarchy is likely to deepen as it becomes sensible to compose multiple types of SCM to balance performance and cost.

The result of these two changes is that there is now a greater burden on system designers to effectively design software to both determine the appropriate sizes and to manage data placement in hierarchical memories. This is especially true in storage systems, where the I/O gap has been especially profound: Fast memories used for caching data have historically been small, because they have been built entirely in RAM. As such, relatively simple heuristics (such as LRU and extensions such as ARC and CAR) could be used to keep a small and obvious set of hot data available for speedy access. The availability of larger fast memories, such as SCM-based caches, moves cached accesses farther out into the “tail” of the access distribution, where both sizing and prediction are much more formidable challenges. Put another way: a storage system has to work a lot harder to get value out of fast memories as it moves further into the tail of an access distribution.

We faced exactly these problems in the design of an enterprise storage system that attempts to balance performance and cost by composing a variety of memory types into a single coherent file system. One challenge we encountered early on involved understanding exactly how much high-performance storage is required to service a given workload. It turns out that while many storage administrators have a good understanding of the raw volume of data they’re dealing with, they’re often at a loss when it comes to predicting how much of that data is hot—and they lack the tools to find out.

AND STORAGE

The Elusive “Working Set”

The concept of a working set is well established within system design [1]. A working set is the subset of a program’s memory that is accessed over a period of execution. Working sets capture the concept of access locality and are the intuition behind the benefits of caching and hierarchical memories in general. A program’s working set is expected to shift over time as it moves between phases of execution or shifts to operate on different data, but the core intuition is that if a program can fit its working set entirely into fast memory, that it will run quickly and efficiently.

While the idea of a working set is relatively simple, it proves to be a very challenging characteristic to measure and model. One aspect of this is that working sets are very different depending on the period of time that they are considered over. A processor architect might consider working set phases to be the sort of thing that distill value from L1 or L2 caches: possibly megabytes of data over several thousand basic blocks of execution. In this domain, working sets may (and do) shift tens or hundreds of times a second. Conversely, a storage system may be concerned with workload characteristics that span minutes, hours, or even days of execution.

A second challenge in characterizing working sets is to measure them at all, at any range in time. Identifying working sets requires tracking the recency of access to addressable memory over time, which is generally both hard and expensive to do. One longstanding approach to this is Mattson’s stack algorithm, which is used to model hit ratio curves (also more pessimistically referred to as miss ratio curves in some of the literature) over an LRU replacement policy.

Mattson’s stack algorithm [4] is a simple technique that provides a really useful result: Given a stream of memory accesses over time, and assuming that those accesses are sent through a cache that is managed using an LRU replacement, Mattson’s algorithm can be run once over the entire trace and will report the hit rate at all sizes of the cache. The algorithm works by maintaining a stack of all addresses that have been accessed and an accompanying array of access counts at each possible depth within that stack. For each access in the stream, the associated address is located in the stack, the counter at that depth is incremented by one, and then that address is pulled to the front of the stack. At the end of the trace, the array is a histogram of reuse distances that directly reports the hit ratio curve. For progressively larger caches, it indicates the number of requests that would have hit in a cache of that size.

The hit ratio curves produced by Mattson (by plotting cache size on the x-axis and hit rate on the y-axis) are a useful way to identify working sets: Horizontal plateaus indicate a range of cache allocation that will not assist workload performance, while sudden jumps in hit rate indicate the edges of working sets, where a specified amount of cache is able to effectively serve a workload.

Unfortunately, calculating HRCs using Mattson is prohibitively expensive, in both time and space, for production systems. Even with optimizations that have been proposed over the decades that the technique has been studied, its memory consumption is linear with the amount of data being addressed, and lookups require log complexity over that set of addresses. This is far too heavyweight to perform at the granularity of every single access. The offline calculation of HRCs is similarly challenging because of the requirement that it carries for trace collection and storage: The resulting I/O traces are very large and challenging to ship to a central point of analysis.

So while modeling working sets has the potential to offer a great deal of insight into storage workloads, especially in regard to managing hierarchical memories, it is too expensive to run in production and so cannot be used for online decisions. Moreover, traces are prohibitively large to ship centrally, making it challenging for system designers to learn from and adapt products to customer workloads. To take full advantage of SCMs in the system, we wanted to achieve both of these things, and so needed a better approach to characterizing working sets.

Counter Stacks

The counter stack [5] is a data structure designed to provide an efficient measure of uniqueness over time. In the case of storage workloads, we are interested in measuring the number of unique block addresses accessed over a window of time. Mattson’s original algorithm (and its subsequent optimizations) measure this by tracking accesses to individual blocks, leading to high memory overheads. Counter stacks have much lower overheads because they do not bother recording accesses to individual addresses, but instead track only the *cardinality* of the accessed addresses. In other words, counter stacks measure *how many* unique blocks are accessed during a given interval, but they do not record the identity of those blocks. By making some relatively simple comparisons of the cardinalities observed over different intervals, we are able to compute approximate reuse distances and, by extension, miss ratio curves.

FILE SYSTEMS AND STORAGE

Counter Stacks and the Elusive Working Set

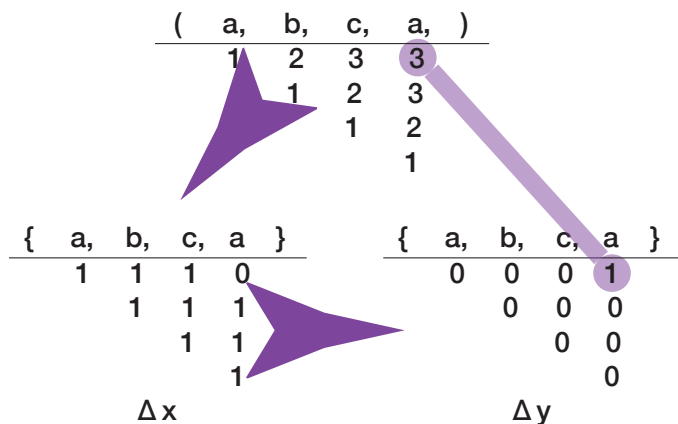


Figure 1: Using cardinality counters to characterize uniqueness over time

To see how this works, consider the sequence of requests for disk addresses $\{a, b, c, a\}$ shown in Figure 1. Imagine that we instantiate a *cardinality counter* for each request we see. Cardinality counters support two operations: *update()* accepts arbitrary 64-bit values, and *count()* returns the total number of unique values passed to *update()*. Cardinality counters can be trivially implemented with a hash table; in practice, *probabilistic counters* like HyperLogLog (see sidebar) use approximation techniques to provide fairly accurate estimates with very low overheads.

In a counter stack, each cardinality counter records the number of unique addresses observed since that counter's inception. For each request, we update every existing counter and also instantiate a new one. If a request increases the value of a given counter, we know that the address has not been accessed at any time since the start of the counter; likewise, if the request does not increase a counter's value, we know that the address must have been previously accessed some time after the start of that counter.

This property makes it easy to pinpoint the logical time at which a requested address was last accessed. For every request, we iterate through the counters from youngest to oldest, updating each as we go. The first counter whose value does not change is necessarily the youngest counter old enough to have observed the last access to the address. Moreover, we know that the last access occurred at exactly the time that this counter was instantiated. If every counter's value changes for a given request, we know that address has never been observed before.

In the example from the diagram, the first matrix gives the values of the counters started for each request. Each row shows the sequence of values for a particular counter, and each column gives the values of the counters at a particular time. We can see that there are four requests for three unique addresses, and at the end of the sequence, each counter has a value of three or less, depending on how many requests it has observed.

We perform two transformations on the matrix to compute reuse distances. First, we calculate Δx , or the difference of

each counter's value with its previous value. Each cell of Δx will have a value of 1 if the counter had not previously observed the request seen at that time, or 0 if it had. Then we calculate Δy , or the difference between adjacent rows of Δx . Each cell of Δy will have a value of 1 if and only if the corresponding request was not previously observed by the younger counter but was observed by the older counter.

A non-zero entry of Δy marks the presence of a repeated address in the request stream, and the row containing such an entry represents the youngest counter to have observed the previous access to the given address. We look up the cell's coordinates in the original matrix to obtain the cardinality value of the corresponding counter at the time of the repeated access, which gives us the number of unique addresses observed since the last access to the given address—in other words, the reuse distance of that address. Similar to Mattson's algorithm, we aggregate these reuse distances in a histogram, which directly gives a miss ratio curve.

Implementing counter stacks with a perfect counter (like a hash table) would be many orders of magnitude more expensive than Mattson's algorithm. Probabilistic counters go a long way towards making this approach feasible in practice, but at roughly 5 KB per counter, maintaining one per request for a workload with billions of requests is still prohibitively expensive. But as the diagram hints, the counter stack matrix is highly redundant and readily compressible.

We employ two additional lossy compression techniques to control the memory overheads of counter stacks. First, instead of maintaining a counter for every request in a workload, we only maintain counters for every k th request, and we only compute counter values after every k th request. This *downsampling* introduces uncertainty proportional to the value k . Second, we periodically *prune* requests as their values converge on those of their predecessors. Convergence occurs when younger counters eventually observe all the same values their predecessors have (it should be clear that counter values will never *diverge*). When the difference in the value of two adjacent counters falls below a pruning distance p , we can reap the younger counter since it provides little to no additional information.

These compression techniques are quite effective in practice, and they provide a means of lowering memory and storage overheads at the cost of reduced accuracy. In our experiments, we have observed that counter stacks require roughly 1200x less memory than Mattson's original algorithm while producing miss ratio curves with mean absolute errors comparable to other approximation techniques that have much higher overheads. Moreover, counter stacks are fast enough to use on the hot path in production deployments: we can process 2.3 million requests per second, compared to about 600,000 requests per second with a highly optimized implementation of Mattson's algorithm.

Probabilistic Counters

Probabilistic counters are a family of data structures that are used to approximate the number of distinct elements within a stream. HyperLogLogs [2] are a common example of such a probabilistic cardinality estimator and have been characterized as allowing cardinalities of over 10⁹ elements to be estimated in a single streaming scan within 2% accuracy using only 1.5 KB of memory. As a result, these estimators are now being used in the implementation of network monitoring, data mining, and database systems [3]. Counter stacks [5] take advantage of HyperLogLogs to efficiently count cardinality in individual epochs during the request stream.

In many senses, HyperLogLogs are a data structure that is similar to, but more restrictive than, Bloom filters. An appropriately sized Bloom filter can provide an accurate hint as to whether or not a specific object has been inserted into it, but does not encode how many objects have been inserted. By simply adding an integer counter, Bloom filters can be extended to estimate cardinality. A HyperLogLog summarizes *just* the total cardinality of distinct objects, and cannot directly answer questions about whether a given object has been inserted. The result

of sacrificing tests of membership is that HyperLogLogs can accurately estimate cardinality with much lower space requirements than would be needed to achieve the same precision using a Bloom filter-based counter.

A detailed explanation of how HyperLogLogs work would require more space than is available here, but the core intuition is relatively simple: If we were to consider a long series of coin tosses, one approach to approximate the total number of flips would be to observe the longest series of consecutive “heads” over the entire stream. Probabilistically, it will take much longer to have 10 heads in a row than it will to have 2; the longest string of heads provides a rough approximation of the total number of tosses. HyperLogLogs work similarly: They hash each element in a stream and then count the number of leading zero bits in the resulting hashed value. By aggregating counts of leading zeros into a set of independently sampled buckets, and then taking the harmonic mean across those resulting independent counts, HyperLogLogs are able to provide a very accurate (significantly better than the coin toss example above) and very compact approximation of the total cardinality.

Strictly speaking, only the last two columns of the counter matrix are needed to compute a miss ratio curve: The values produced by computing Δx and Δy can be incrementally aggregated into a histogram as the algorithm works through the sequence of requests, and older columns can be discarded. However, the matrix provides a convenient record of workload history, and, with a simple transformation (amounting in essence to a column index shift), it can be used to compute miss ratio curves over arbitrary sub-intervals of a given workload. This functionality turns out to be very expensive with traditional techniques for computing miss ratio curves, but it can be quite useful for tasks like identifying workload anomalies and phase changes.

In fact, we’ve found that counter stacks can help to answer a number of questions that extend beyond the original problems that led us to develop them. In particular, they provide an extremely concise format for preserving workload histories in the wild. We use counter stacks to record and transfer access patterns in production deployments at the cost of only a few MB per month; the next best compression technique we evaluated had a roughly 50x overhead. The ability to retain extended workload histories—and ship them back for easy analysis—is invaluable for diagnosing performance problems and understanding how our system is used in general, and it is enabling a new data-driven approach to designing placement algorithms. As we learn more about real-world workloads, we expect to augment counter stacks with additional metadata, thereby providing a richer representation of application behavior.

References

- [1] Peter J. Denning, “The Working Set Model for Program Behavior,” *Communications of the ACM*, vol. 11, no. 5 (May 1968), pp. 323–333.
- [2] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, “HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm,” in *Proceedings of the 2007 International Conference on Analysis of Algorithms (DMTCS, 2007)*, pp. 127–146.
- [3] S. Heule, M. Nunkesser, and A. Hall, “HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm,” in *Proceedings of the 16th International Conference on Extending Database Technology (EDBT ’13) (ACM, 2013)*, pp. 683–692.
- [4] R. L. Mattson, J. Gecsei, J. D. R. Slutz, and I. L. Traiger, “Evaluation Techniques for Storage Hierarchies,” *IBM-Systems Journal*, vol. 9, no. 2 (1970), pp. 78–117.
- [5] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, “Characterizing Storage Workloads with Counter Stacks,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI ’14) (USENIX Association, 2014)*, pp. 335–349.

Storage Options for Software Containers

MARK LAMOURINE



Mark Lamourine is a senior software developer at Red Hat. He's worked for the last few years on the OpenShift project. He's a coder by training, a

sysadmin and toolsmith by trade, and an advocate for the use of Raspberry Pi style computers to teach computing and system administration in schools. Mark has been a frequent contributor to the *;/login:* book reviews. markllama@gmail.com

Software containers are likely to become a very important tool over the next few years. While there is room to argue whether or not they are a new tool, it is clear that they have certain elements that are clearly immature. Storage is one of those elements.

The problem isn't that we need new storage services. The new factors are due to the characteristics of containers themselves and how they differ from traditional bare-metal hosts and virtual machines (VMs). Also, storage isn't an issue on single hosts where it can be mounted manually for each individual container. Large container farms present problems that are related to those for VM-based IaaS services but that are complicated by VMs' lack of clean boundaries.

There are two common container mechanisms in use today: LXC and Docker. LXC is the older mechanism and requires careful crafting of the container environment, although it also provides more control to the user. Creating LXC containers requires a significant level of expertise. LXC also does not provide a simple means to copy and re-instantiate existing containers.

Docker is the more recent container environment for Linux. It makes a set of simplifying assumptions and provides tools and techniques that make creating, publishing, and consuming containers much easier than it has ever been. This has made container technology much more appealing than it was before, but current container systems only manage individual containers on single hosts. As people begin trying to put containers and containerized applications to use at larger scales, the remaining problems, such as how to manage storage for containers, are exposed.

In this article I'm going to use Docker as the model container system, but all of the observations apply as well to LXC and to container systems in general.

A Container Primer

The first thing to understand is that containers don't contain [1]. A container is really a special view of the host operating system that is imposed on one or more processes. The "container" is really the definition of the view that the processes will see. In some senses they are similar to *chroot* environments or *BSD jails* but the resemblance is superficial and the mechanism is entirely different.

The enabling mechanism for Linux containers is *kernel namespaces* [2, 3]. Kernel namespaces allow the kernel to offer each process a different view of the host operating system. For example, if a contained process asks for `stat(3)` for the root (`/`), the namespace will map that to a different path (when seen by an uncontained process): for example, `/var/lib/docker/devicemapper/mnt/<ID>/rootfs/`. Since the file system is hierarchical, requests for information about files beneath the root node will return answers from inside the mapped file tree.

In all *NIX operating systems, PID 1 is special. It's the init process that is the parent of all other processes on a host. In a Docker container, there is a master process, but it is generally not the system process. Rather, it may be a shell or a Web server. But from inside the container, the master process will appear to have PID 1.

There are six namespaces that allow the mapping of different process resources [6]:

- ◆ mount—file systems
- ◆ UTS—nodename and domain name
- ◆ IPC—inter-process communication
- ◆ PID—process identification
- ◆ network—network isolation
- ◆ user—UID mapping

A process running “in a container” is, in fact, running directly on the container host. All of the files it sees are actually files inside the host file system. The “containment” of the process is an illusion, but a useful one. This lack of the traditional boundaries is what makes container storage management something new.

Software Container Farms and Orchestration

If all you want to do is run a single container on a single host with some kind of storage imported, there's no real problem. You manually create or mount the storage you want, then import the storage when you create the container. Both LXC and Docker have means of indicating that some external file-system root should be re-mapped to a new mount point inside the container. When you want to create a container farm, where placement of individual containers is up to the orchestration system, then storage location becomes interesting. In a container farm, the person who requests a new container only gets to specify the characteristics, not the deployment location.

There are a number of container orchestration systems currently under development. CoreOS is using a system called Fleet [3]. Google and Red Hat are working on Kubernetes [4]. Both have slightly different focus and features but in the end they will both have to create the environment necessary to run containers on hosts that are members of a cluster. I think it's too early to tell what will happen in the area of container orchestration system development even over the short term.

I'm not going to talk about how the orchestration systems will do their work, I'm only going to talk about the flavors of storage they will be called on to manage and the characteristics and implications of each. But first, let's look at how Docker handles storage without an orchestration system.

Docker Ephemeral Storage

When you ask Docker to create a container, it unpacks a collection of tar archives that together are known as the *image*. If no external volumes are specified, then the only file tree mounted is the unpacked image. Each of the running containers is unpacked into `/var/lib/docker/devicemapper/mnt/<ID>/rootfs` where `<ID>` is the container ID returned when the container is created using the devicemapper driver. Different storage drivers will have slightly different paths.

This is *ephemeral storage* in the sense that when the container is deleted, the storage is reclaimed and the contents deleted (or unmounted). This file tree is not shared with other containers. Changes made by processes in the container are not visible to other containers.

Docker Volumes—Shared and Persistent Storage

The Dockerfile VOLUME directive is used to define a mount point in a container. It basically declares, “I may want to mount something here or share something from here.” Docker can mount different things at that point in response to arguments when a container is created.

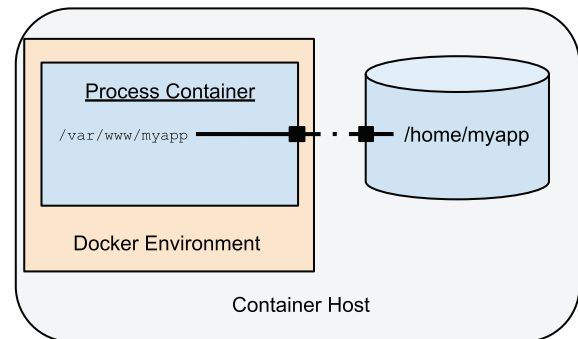


Figure 1: The Docker VOLUME directive creates a mount point in the container. The file tree below the mount point in the image is placed in a separate space when the container is created. It can be exported to other containers or imported either from a container or from external storage.

When you create a new Docker container from an image that declares a volume, but you don't provide an external mount point, then the content below the volume mount point is placed in its own directory within the Docker workspace (`/var/lib/docker/vfs/dir/`).

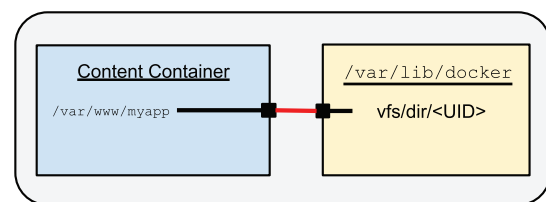


Figure 2: A container connected to an “internal” volume. This is created by Docker as a default if no external volume is offered.

FILE SYSTEMS AND STORAGE

Storage Options for Software Containers

You can share this storage between containers on the same host with the `docker --volumes-from` command line option. This causes the declared volumes from an existing container to be mounted on any matching volumes in the new container.

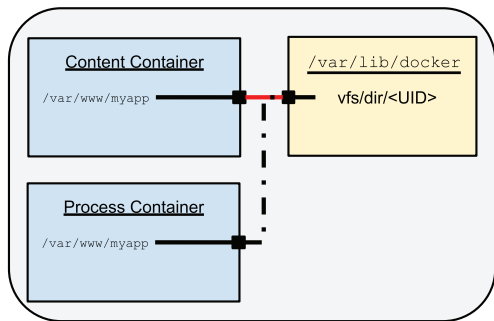


Figure 3: Two containers sharing a volume. The volume is created by Docker when the first container is created. The second container mounts from the first using the `--volumes-from` option.

Shared storage using what's known as a *data container* can be treated as persistent across restarts of an application. The application container can be stopped and removed and then recreated or even upgraded. The data container will be available to remount when the application is restarted.

This volume sharing will also work with *host storage*.

External Host Storage

In this case “external” means “from outside the container.” When you start a Docker container, you can indicate that you want to mount a file or directory from the host to a point inside the container.

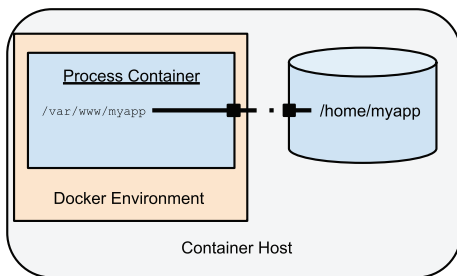


Figure 4: A container with host storage. The host storage is bind mounted onto the container volume mount point.

Host storage is useful when you are creating individual containers on a single host. You can create and manage the space on the host that you're going to mount inside before you start the container. This gives you more control over the initialization of the storage, and you can view and modify it easily from the host while the application is running.

This becomes more difficult, though, when you start working with a large container farm. The whole idea of container farms is that all of the hosts are identical and that the containers can be placed and moved to maintain the balance of the load within the cluster. The only way to do that practically is to move the storage off the container host entirely.

Containers, the `mount(8)` Command and Container Orchestration

I've mentioned the lack of the host boundary when managing containers. The `mount(8)` command is where this appears for storage. You can't run `mount(8)` from inside a container without special privileges. Since a container is just a special view of the host, any file system mounted into a container must be mounted onto the host before the container is started. In general, processes inside containers are not given the privileges to affect anything on the host outside the container. (If they can, they're not very well contained, are they?)

For similar reasons, Docker cannot cause the host to mount new file systems, whether local or remote. Docker restricts itself to controlling how the container sees the resources that the host already has available. Docker manages individual containers on single hosts. For large numbers of containers spread across multiple hosts, the orchestration system will have to provide a way to connect storage to containers within the cluster. In the rest of this article, I'll talk about where the storage will come from.

Docker and the Host Boundary

At this point you've seen everything that Docker can do with storage. Docker limits itself to using the resources available on a host. Its job is to provide those resources to the interior of containers while maintaining the limited view the containers have of the host outside. This means that Docker itself is unaware of any containers on other hosts or of any other resource that has not been attached to the host when the container is created. Docker can't make changes to the host environment on behalf of a container.

This is where orchestration systems come in. A good orchestration system will have a way to describe a set of containers and their interactions to form a complete application. It will have the knowledge and means to modify the host environment for new containers as they are created.

Network Storage

Most machines have block storage mounted directly on the host. Network storage extends the reach of individual hosts to a larger space than is possible with direct storage, and it allows for the possibility of sharing storage between multiple hosts.

I'm going to group all of the traditional off-host storage services under the umbrella of "network storage." These include NAS, SAN, and more modern network storage services. There are a few differences.

NAS services like NFS, CIFS, and AFS don't appear on the host as devices. They operate using IP protocols on the same networks that carry the other host traffic. Their unit of storage is a file (or directory). They generally don't tolerate latency very well. In their original form, NAS services don't support distributed files or replication. In most cases they don't require a service running on the client host to manage the creation of connections or the file traffic. NAS services can be provided by specialized appliances or by ordinary hosts running the service software.

There is a new class of distributed network services that provide replication and higher performance than single-point NAS does. Gluster and Ceph are two leading distributed NAS services. Clients run special daemons that distribute and replicate copies of the files across the entire cluster. The files can either be accessed on the client hosts or be served out over NFS to other clients.

SAN systems include Fibre Channel, InfiniBand, and iSCSI. Fibre Channel and InfiniBand require special hardware networks and connections to the host. iSCSI can run over IP networks and so does not require special hardware and networks, although for the best performance, users often need to create distinct IP networks to avoid conflicts with other data traffic. SAN services usually require some additional software to map the service end points to *NIX device files, which can be partitioned, formatted, and mounted like ordinary attached storage. SAN services provide very low latency and high throughput, to the point where they can substitute for attached storage.

For container systems these all pose essentially the same problem. The orchestration system must locate and mount the file system on the host. Then it must be able to import the storage into the container when it is created.

File Ownership and UID Mapping

One major unsolved problem for Docker (at the time of this writing) is control of the ownership and access permissions on files.

*NIX file ownership is defined by UID and GID numbers. For a process to access a file, the UID of a process must match the file owner UID, and the user must have the correct permissions or get access via group membership and permissions. With the exception of the root user (UID 0, GID 0), the assignment of UID/GID is arbitrary and by convention.

The UID assignment inside a container is defined by the `/etc/passwd` file built into the container image. There's no relation to the UID assignment on the container host or on any network storage.

When a process inside a container creates a file, it will be owned by the UID of the process in the container even when seen from the host. When using host, network, or cloud block storage, any process on the host with the same UID will have the same access as the processes inside the container.

Access in the other direction is also a problem. If files on shared storage are created with a UID that does not match the process UID inside the container, then the container process will fail when accessing the storage.

This will also benefit developers trying to create generic container images that are able to share storage between containers. Currently, any two containers that mean to share storage must have identical user maps.

The Linux kernel namespace system includes the ability to map users from inside a container to a different one on the host. The Docker developers are working on including user namespaces, but they present a number of security issues that have to be resolved in the process.

Container Hosts and Storage Drivers

Even before the introduction of Docker there was a movement to redefine the way in which software is packaged and delivered. CoreOS [5] and, more recently, Project Atomic [6] are projects which aim to create a stripped down host image that contains just the components needed to run container applications. Since they just run containers, much of the general purpose software normally installed on a host or VM isn't needed. These lean images do not need patching. Rather, the host image is replaced and rebooted as a unit (hence, "Atomic").

Although this simplifies the maintenance of both the host and the containers, using "forklift updates," the rigid image formats make adding drivers or other customizations difficult. There is a very strong pressure to keep the host images small and to include only critical software. Everything that can be put into a container is, even tools used to monitor and manage the container host itself.

These purpose-made container hosts will need to provide a full range of network storage drivers built into the image, or they will have to be able to accept drivers running in containers if they want to compete with general purpose hosts configured for containers. It's not clear yet which drivers will be available for these systems, but they are being actively developed.

Cloud Storage

Cloud services take a step further back. They disassociate the different components of a computer system and make them self-service. They can be managed through a user interface or through an API.

FILE SYSTEMS AND STORAGE

Storage Options for Software Containers

Cloud storage for applications usually takes one of two forms: *object storage* and *block storage*. (The third form of cloud storage, *image storage*, is used to provide bootable devices for VMs.)

Object Storage

All of the cloud services, public and private, offer some form of object storage, called *Swift* in OpenStack. The AWS object store is *S3*, and Google offers *Google Cloud Storage* (not to be confused with Google Cloud Engine Storage; see “Block Storage,” below).

Object stores are different from the other storage systems. Rather than mounting a file system, the data are retrieved through a REST API directly by processes inside the container. Each file is retrieved as a unit and is placed by the calling application into an accessible file space. This means that object storage doesn’t need any special treatment by either the container system or the orchestration system.

Object stores do require some form of identification and authentication to set and retrieve data objects. Managing sensitive information in container systems is another area of current work.

Container images that want to use object stores must include any required access software. This may be an API library for a scripting language or, possibly, direct coding of the HTTP calls.

The push-pull nature of object stores makes them unsuitable for uses that require fast read/write access to small fragments of a file. Access can have very high latency, but the objects are accessed as a unit, so they are unlikely to be corrupted during the read/write operations. The most common uses are for configuration files and for situations where data inconsistency from access collisions can be accepted in the short term.

Block Storage

Cloud block storage appears on a (virtual) host as if it were direct attached storage. Each cloud storage system has a different name for its own form of block storage. OpenStack uses the *Cinder* service. On Amazon Web Services it’s known as *EBS*. Google calls it *GCE Storage* (not to be confused with Google Cloud Storage).

Cloud block storage systems are controlled through a published API. When a process requests access to cloud storage, a new device file is created. Then the host can mount the device into the file system. From there Docker can import the storage into containers.

The challenge for an orchestration system is to mount each block device onto a container host on-demand and make it available to the container system. Since each cloud environment has a different API, either they all must be hard-coded into the orchestration system or the orchestration system must provide a plugin mechanism.

So far the only combination I’ve seen work is Kubernetes in Google Cloud Engine, but developers on Kubernetes and others all recognize the need for this feature and are actively developing.

Summary

Container systems in general and Docker in particular are limited in scope to the host on which they run. They create containers by altering the view of the host that contained processes can see. They can only manage the resources that already exist on the host.

Orchestration systems manage multiple containers across a cluster of container hosts. They allow users to define complex applications composed of multiple containers. They must create or configure the resources that the containers need, and then trigger the container system, like Docker, to create the containers and bind them to the resources.

Currently, only Kubernetes can mount GCE Storage when running in the GCE environment.

For container systems to scale, the orchestration systems will need to be extended to be able to communicate and manage the various network and cloud storage systems. Docker and the orchestration systems will need to be able to manage user mapping as well as file access controls.

In both Fleet and Kubernetes, the development teams are actively working to address all of these issues, and I expect that there will be ways to manage storage in large container farms very soon. Once there are, containers will begin to fulfill their promise.

For a more detailed treatment of containers, see the article by Bottomley and Emelyanov [7].

References

- [1] Daniel Walsh, “Are Docker Containers Really Secure?": <http://opensource.com/business/14/7/docker-security-selinux>.
- [2] Linux kernel namespaces: <http://lwn.net/Articles/531114/>.
- [3] CoreOS Fleet: <https://github.com/coreos/fleet>.
- [4] Google Kubernetes: <https://github.com/GoogleCloudPlatform/kubernetes>.
- [5] CoreOS container hosts: <https://coreos.com/>.
- [6] Project Atomic: <http://www.projectatomic.io/>.
- [7] James Bottomley and Pavel Emelyanov, “Containers,” *login.*, vol. 39, no. 5, Oct. 2014: <https://www.usenix.org/publications/login/october-2014-vol-39-no-5/containers>.

Interview with Steve Swanson

RIK FARROW



Steven Swanson is an associate professor in the Department of Computer Science and Engineering at the University of California, San Diego and

the director of the Non-Volatile Systems Laboratory. His research interests include the systems, architecture, security, and reliability issues surrounding non-volatile, solid-state memories. He also co-leads projects to develop low-power co-processors for irregular applications and to devise software techniques for using multiple processors to speed up single-threaded computations. In previous lives he has worked on scalable dataflow architectures, ubiquitous computing, and simultaneous multithreading. He received his PhD from the University of Washington in 2006 and his undergraduate degree from the University of Puget Sound.

swanson@eng.ucsd.edu



Rik is the editor of *login*.
rik@usenix.org

While walking the poster session at OSDI '14, I saw Steve Swanson. I wanted to talk to him about the past and future of non-volatile memory (NVM), since he and his students had produced many papers about the topic that I'd heard presented. I also had some vague ideas about a class for FAST '15 on the history of NVM. Alas, Steve was moving so quickly through the crowd that I never caught up to him in Broomfield.

However, Steve later agreed to an email interview.

Rik: How did you get interested in non-volatile storage?

Steve: I was a new professor and was talking to a colleague from industry who was working on storage systems. He mentioned that flash memory was really beginning to shake things up, so I took a look at it. It turned out to be a perfect research problem for me: It involved hardware and system design (which is, broadly, what I did my PhD on), and it centered around a huge disruption in the performance of a particular piece of the system. Those kinds of changes always result in big challenges and open up all kinds of new areas for research. My students and I dove in, and it's been really exciting and has allowed us to do interesting work on both the hardware and software side as well as at the application level.

Rik: I wanted to start off with some history, or at least try to better understand how we got to where we are today with flash-based storage devices. Having heard many paper presentations, it seems like there have been, and will continue to be, two big issues, both of them interrelated.

These are the flash translation layer (FTL) and the disk-style interface for flash-based storage. Can you explain why vendors adopted these interfaces?

Steve: FTLs arose because SSD vendors needed to make it as easy as possible for customers to use their new drives. It's a much simpler proposition to sell something as a faster, drop-in replacement for a hard drive. If you can make your flash drive look like a hard drive, you immediately have support from all major operating systems, you can use existing file systems, etc. The alternative is to tell a customer that you have a new, fast storage device, but it will require them to completely change the way their software interacts with storage. That's just a non-starter.

The disk-based interface that FTLs emulate emerged because it is a natural and reasonably efficient interface for talking to a disk drive. Indeed, just about everything about how software interacts with storage has been built up around disk-based storage. It shows up throughout the standard file-based interfaces that programmers use all the time.

The problem is that flash memory looks nothing like a disk. The most problematic difference is that flash memory does not support in-place update. Inside an SSD, there are several flash chips. Each flash chip is broken up into 1000s of "blocks" that are a few hundred kilobytes in size. The blocks are, in turn, broken into pages that are between 2 and 16 KB.

FILE SYSTEMS AND STORAGE

Interview with Steve Swanson

Flash supports three main operations. First, you can “erase” a block, that is, set it to all 1s. It seems like “erased” should be all 0s but the convention is that it’s all 1s. Erasing a block takes a few milliseconds. Second, you can “program” a page in an erased block, which means you can change some of the 1s to 0s. You have to program the whole page at once, and you must program the pages within a block in order. Programming takes hundreds of microseconds. Third, you can read a page, and reading takes tens of microseconds. The result of this is that if you want to change a value in a particular page, you need to first erase the entire block and then reprogram the entire block. This is enormously inefficient.

The final wrinkle is that you can only erase each block a relatively small number of times before it will become unreliable—between 500 and 100,000 depending on the type of flash chip. This means that even if erasing and reprogramming a block were an efficient way to modify flash, performing an erase on every modification of data would quickly wear out your flash.

So the FTL’s job is pretty daunting: It has to hide the asymmetry between programs and erasures, ensure that erasures are spread out relatively evenly across all the flash in the system so that “hot spots” don’t cause a portion of the flash to wear out too soon, present a disk-like interface, and do all this efficiently and quickly. Meeting these challenges has turned out to be pretty difficult, but SSD manufacturers have done a remarkably good job of producing fast, reliable SSDs.

The first SSDs looked exactly like small hard drives. They were the same shape, and they connected to the computer via standard hard drive interface protocols (i.e., SATA or SAS). But those protocols were built for disks. Flash memory provided the possibility of building much faster (in terms of both bandwidth and latency) storage devices than SATA or SAS could support. Importantly, SSD could also support much more concurrency than hard drives, and they supported vastly more efficient random accesses than hard drives.

The first company to take a crack at something better was FusionIO. They announced and demonstrated their ioDrive product in September 2007. Instead of using a conventional form factor and protocol, the ioDrive was a PCIe card (like a graphics card) and used a customized interface that was tuned for flash-based storage rather than disk-based storage. FusionIO also began to experiment with new interfaces for storage, making it look quite different from a disk. It’s not clear how successful this has been. The disk-like interface has a heavy incumbent advantage.

More recently, NVMe Express has emerged as a standard for communicating with the PCIe-attached SSDs. It supports lots of concurrency and is built for low-latency, high-bandwidth drives. Many vendors sell (or will sell shortly) NVMe drives.

Another set of systems has taken a different approach. Rather than use NVMe to connect an SSD to a single system, they build large boxes full of flash and expose them over a network-like interconnect (usually Fibre Channel or iSCSI) to many servers. These network-attached storage (NAS) SSDs must solve all the same problems NVMe or SATA SSDs must solve, but they do address one puzzle that faces companies building high-end SSDs: These new drives can deliver so much storage performance that it’s hard for a single server to keep up. By exposing one drive to many machines, NAS SSDs don’t have that problem. Violin and Texas Memory Systems fall into this camp.

Rik: If vendors have done such a great job with flash, why has there been so much academic research on it?

Steve: I think the main problem here is that most researchers don’t know what industry is actually doing. The inner workings of a company’s FTL are their crown jewels. Physically building an SSD (i.e., assembling some flash chips next to a flash controller on PCB) is not technically challenging. The real challenge is in managing the flash so that performance is consistent and managing errors so that they can meet or exceed the endurance ratings provided by flash chip manufacturers. As a result, the research community has very little visibility into what’s actually going on inside companies. Some researchers may know, but the information is hidden behind NDAs.

Of course, designing a good FTL is an interesting problem, and there are many different approaches to take, so researchers write papers about them. However, it’s not clear how much impact they will have. Maybe the techniques they are proposing are cutting edge, extend the state of the art, and/or are adopted by companies. Or maybe they aren’t. It’s hard to tell, since companies don’t disclose how their FTLs work.

My personal opinion is that, on the basic nuts and bolts of managing flash, the companies are probably ahead of the researchers, since that technology is directly marketable, the companies are better funded, and they have better access to proprietary information about flash chips, etc.

I think researchers have the upper hand in terms of rethinking how SSD should appear to the rest of the system—for example, adding programmability or getting away from the legacy block-based interface, since this kind of fundamental rethinking of how storage should work is more challenging in the commercial environment. However, I think it’s probably the more interesting part of SSD research and, in the long term, will have more impact than, for example, a new proprietary wear-leveling scheme.

Rik: I’ve heard several paper presentations that cover aspects of NVMe when it has become byte addressable, instead of block addressable, as it is today. That’s assuming, of course, that the promises come true. Can you talk about future directions for research?

Steve: I think the most pressing questions going forward lie along four different lines:

Byte-addressable memories will probably first appear in small quantities in flash-based SSD. One important question is how can we use small amounts of byte-addressable NVM to improve the performance of flash-based SSDs. This is the nearest-term question, and there are already some answers out there. For instance, it's widely known that FusionIO (now SanDisk) uses a form of byte-addressable NVM in its SSDs.

A second likely early application for NVM is in smartphones and other mobile devices. You can imagine a system with a single kind of memory that would serve the role of modern DRAM and also serve as persistent data storage. Since it would have the performance of DRAM, it could alter the programming model for apps: Rather than interacting with key-value stores and other rather clumsy interfaces to persistent storage, they could just create data structures in persistent memory. This would, I think, be a nice fit for lots of small, one-off apps. The main challenge here is in making it easy for programmers to get the persistent data structures right. It's very hard to program a linked list or tree so that, if power fails at an inopportune moment, you can ensure that the data structure remains in a usable state. We have done some work in this area recently as has Mike Swift's group at the University of Wisconsin in Madison, but there's much left to do.

If we solve the next problem, then many of the techniques that we could use in mobile systems would be applicable in larger systems too.

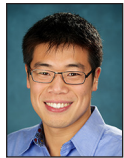
Third, if byte-addressable memories are going to be useful in datacenter-scale storage systems, the data they hold must be replicated, so that if the server hosting one copy goes down, the data is still available. This is a challenge because the memories have access times on the order of tens of nanoseconds, while network latencies are on the order of (at least) a few microseconds. How can we transmit updates to the backup copy without squandering the performance of these new, fast, byte-addressable memories? There are many possible solutions. We've done work on a software-based solution, but it's also possible that we should integrate the network directly into the memory system. This also raises the question of how to reconcile the large body of work from the distributed systems community on distributed replication with the equally large body of work from the architecture community on how to build scalable memory systems. Both fields deal with issues of data consistency and how updates at different nodes should interact with one another, but they do so in very different ways.

The fourth area of interest is in how we can integrate I/O more deeply into programming languages. In modern languages, I/O is an afterthought, so the compiler really has no idea I/O is going on and can't do anything to optimize it. This was not a big deal for disk-based systems, since disk I/O operates on time scales so large (that is, they are so slow) that the compiler could not hope to do anything to improve I/O performance. As storage performance increases, it becomes very feasible that a compiler could, for example, identify I/O operations and execute them a few microseconds early so that the code that needs the results would not have to wait for them. Doing this means we need to formalize the semantics of I/O in a precise way that a compiler could deal with.

Simple Testing Can Prevent Most Critical Failures

An Analysis of Production Failures in Distributed Data-Intensive Systems

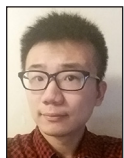
DING YUAN, YU LUO, XIN ZHUANG, GUILHERME RENNA RODRIGUES, XU ZHAO, YONGLE ZHANG, PRANAY U. JAIN, AND MICHAEL STUMM



Ding Yuan is an assistant professor in the Electrical and Computer Engineering Department of the University of Toronto. He works in computer systems, with a focus on their reliability and performance. yuan@ece.toronto.edu



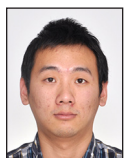
Yu (Jack) Luo is an undergraduate student at the University of Toronto studying computer engineering. He has interned at IBM working with memory management and disaster recovery. His research interests are in systems, failure recovery, and log analysis. jack.luo@mail.utoronto.ca



Xin Zhuang is an undergraduate at the University of Toronto studying computer engineering. His research interest is in software systems. xin.zhuang@mail.utoronto.ca



Guilherme Renna Rodrigues is an exchange student at the University of Toronto and is an undergraduate at CEFET-MG, Belo Horizonte, Brazil. The research project at the University of Toronto has led to his interest in practical solutions for computing systems problems. guilhermerenna@gmail.com



Xu Zhao is a graduate student at the University of Toronto. He received a B.Eng. in computer science from Tsinghua University, China. At U of T, his research is focused on reliability and performance of distributed systems. nuk.zhao@mail.utoronto.ca

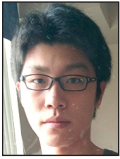
Large, production-quality distributed systems still fail periodically, sometimes catastrophically where most or all users experience an outage or data loss. Conventional wisdom has it that these failures can only manifest themselves on large production clusters and are extremely difficult to prevent a priori, because these systems are designed to be fault tolerant and are well-tested. By investigating 198 user-reported failures that occurred on production-quality distributed systems, we found that almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors, and, surprisingly, many of them are caused by trivial mistakes such as error handlers that are empty or that contain expressions like “FIXME” or “TODO” in the comments. We therefore developed a simple static checker, Aspirator, capable of locating trivial bugs in error handlers; it found 143 new bugs and bad practices that have been fixed or confirmed by the developers.

Our study also includes a number of additional observations that may be helpful in improving testing and debugging strategies. We found that from a testing point of view, almost all failures require only three or fewer nodes to reproduce, which is good news considering that these services typically run on a very large number of nodes. In addition, we found that a majority of the failures can simply be reproduced by unit tests even though conventional wisdom has it that failures that occur on a distributed system in production are extremely hard to reproduce offline. Nevertheless, we found the failure manifestations are generally complex, typically requiring multiple input events occurring in a specific order.

The 198 randomly sampled, real world, user-reported failures we studied are from the issue tracking databases of five popular distributed data-analytic and storage systems: Cassandra, HBase, HDFS, Hadoop MapReduce, and Redis. We focused on distributed, data-intensive systems because they are the building blocks of many Internet software services, and we selected the five systems because they are widely used and are considered production quality.

Software	Language	Failures		
		Total	Sampled	Catastrophic
Cassandra	Java	3,923	40	2
HBase	Java	5,804	41	21
HDFS	Java	2,828	41	9
MapReduce	Java	3,469	38	8
Redis	C	1,192	38	8
Total	-	17,216	198	48

Table 1: Number of reported and sampled failures for the systems we studied, and the catastrophic ones from the sample set



Yongle Zhang is a graduate student in computer engineering at the University of Toronto. His research interests are in operating systems and log analysis. He previously received an M.E. and B.E. in computer science from the Institute of Computing Technology of the Chinese Academy of Sciences and at Shandong University, respectively. yongle.zhang@mail.utoronto.ca



Pranay U. Jain is a final year undergraduate in computer engineering at the University of Toronto. Previously, he interned with the Amazon Web Services team. pranay.ujain@mail.utoronto.ca



Michael Stumm is a professor in the Electrical and Computer Engineering Department of the University of Toronto. He works in the general area of computer systems software with an emphasis on operating systems for distributed systems and multiprocessors. He co-founded two companies, SOMA Networks and OANDA, a currency trading company. stumm@ece.utoronto.ca

Table 1 shows the distribution of the failure sets. For each sampled failure ticket, we carefully studied the failure report, the discussion between users and developers, related error logs, the source code, and patches to understand the root cause and its propagation leading to the failure.

We further studied the characteristics of a specific subset of failures—the *catastrophic failures*, which we define as those failures that affect all or a majority of users instead of only a subset of users. Catastrophic failures are of particular interest because they are the most costly ones for the service providers, and they are not supposed to occur, considering these distributed systems are designed to withstand and automatically recover from component failures.

General Findings

What follows is a list of all of our general findings. Overall, our findings indicate that the failures are relatively complex, but they identify a number of opportunities for improved testing and diagnosis. Note that we only discuss the first five of the general findings in this article. Our OSDI paper [6] contains detailed discussions on the other general findings, and findings for catastrophic failures are discussed below (Findings 11-13).

1. A majority (77%) of the failures require more than one input event to manifest.
2. A significant number (38%) of failures require input events that typically occur only on long running systems.
3. The specific order of events is important in 88% of the failures that require multiple input events.
4. Twenty-six percent of the failures are non-deterministic—they are not guaranteed to manifest given the right input event sequences.
5. Almost all (98%) of the failures are guaranteed to manifest on no more than three nodes.
6. Among the non-deterministic failures, 53% have timing constraints only on the input events.
7. Seventy-six percent of the failures print explicit failure-related error messages.
8. For a majority (84%) of the failures, all of their triggering events are logged.
9. Logs are noisy: the median of the number of log messages printed by each failure is 824.
10. A majority (77%) of the production failures can simply be reproduced by a unit test.

Finding 1: *A majority (77%) of the failures require more than one input event to manifest, but most of the failures (90%) require no more than three.*

Figure 1 provides an example where two input events, a load balance event and a node crash, are required to take down the cluster. Note that we consider the events to be “input events” from a testing and diagnostic point of view—some of the events (e.g., “load balance” and “node crash”) are not strictly user inputs but can easily be emulated in testing.

Finding 2: *A significant number (38%) of failures require input events that typically occur only on long running systems.*

The load balance event in Figure 1 is such an example. This finding suggests that many of these failures can be hard to expose during normal testing unless such events are intentionally exercised by testing tools.

Finding 3: *The specific order of events is important in 88% of the failures that require multiple input events.*

Consider again the example shown in Figure 1. The failure only manifests when the load balance event occurs before the crash of slave B. A different event order will not lead to failure.

Simple Testing Can Prevent Most Critical Failures

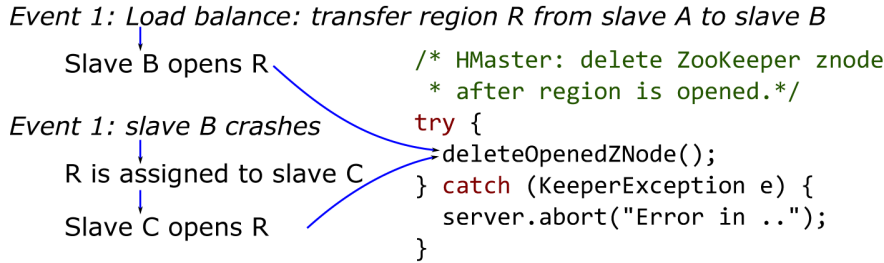


Figure 1: A failure in HBase that requires two input events to trigger. A load balance event first causes a region R to be transferred from an overloaded slave A to a more idle slave B. After B opens R, HMaster deletes the ZooKeeper znode that is used to indicate R is being opened. If slave B crashes at this moment, another slave C is assigned to serve the region. After C opens R, HMaster tries to delete the same ZooKeeper znode again, but deleteOpenedZNode() throws an exception because the znode is already deleted. This exception takes down the entire cluster.

In many cases, even with the right combination and sequence of input events the failure is not guaranteed to manifest:

Finding 4: *Twenty-six percent of the failures are non-deterministic—they are not guaranteed to manifest given the right input event sequences.*

In these cases, additional timing relationships are required for the failures to manifest. For example, the failure in Figure 1 can only manifest when slave B crashes after the znode is deleted. If it crashes before the HMaster deletes the znode, the failure would not be triggered.

Findings 1–4 show the complexity of failures in large distributed systems. To expose the failures in testing, we need to not only explore the combination of multiple input events from an exceedingly large event space with many only occurring on long running systems, we also need to explore different permutations. Some further require additional timing relationships.

The production failures we studied typically manifested themselves on configurations with a large number of nodes. This raises the question of how many nodes are required for an effective testing and debugging system.

Finding 5: *Almost all (98%) of the failures are guaranteed to manifest on no more than three nodes.*

The number is similar for catastrophic failures, where 98% of them manifest on no more than three nodes. Finding 5 implies that it is not necessary to have a large cluster to test for and reproduce failures.

Note that Finding 5 does not contradict the conventional wisdom that distributed system failures are more likely to manifest on large clusters. In the end, testing is a probabilistic exercise. A large cluster usually involves more diverse workloads and fault modes, thus increasing the chances for failures to manifest. However, what

our finding suggests is that it is not necessary to have a large cluster of machines to expose bugs, as long as the specific sequence of input events occurs.

Catastrophic Failures

Table 1 shows that 48 failures in our failure set have catastrophic consequences. We classify a failure to be catastrophic when it prevents all or a majority of the users from their normal access to the system. In practice, these failures result in a cluster-wide outage, a hung cluster, or a loss to all or to a majority of the user data.

The fact that there are so many catastrophic failures is perhaps surprising given that the systems considered all have high availability (HA) mechanisms designed to prevent component failures from taking down the entire service. For example, all of the four systems with a master-slave design—namely, HBase, HDFS, MapReduce, and Redis—are designed to, on a master node failure, automatically elect a new master node and fail over to it. Cassandra is a peer-to-peer system and thus by design avoids single points of failure. Then why do catastrophic failures still occur?

Finding 11: *Almost all catastrophic failures (92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software (see Figure 2).*

These catastrophic failures are the result of more than one fault triggering, where the initial fault, whether due to hardware, misconfiguration, or bug, first manifests itself explicitly as a non-fatal error—for example, by throwing an exception or having a system call return an error. This error need not be catastrophic; however, in the vast majority of cases, the handling of the explicit error was faulty, resulting in an error manifesting itself as a catastrophic failure.

Overall, we found that the developers are good at anticipating possible errors. In all but one case, the errors were properly checked for in the software. However, we found the developers were often negligent in handling these errors. This is further

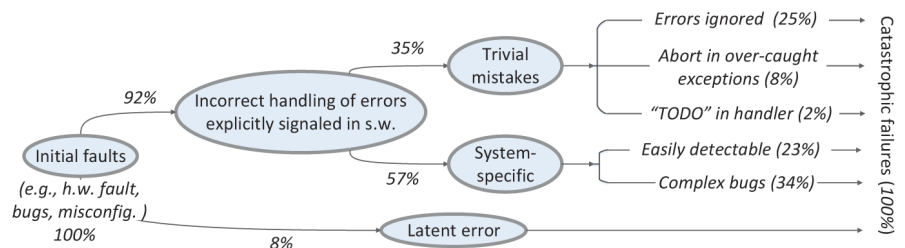


Figure 2: Breakdown of all catastrophic failures by their error handling

Simple Testing Can Prevent Most Critical Failures

corroborated in Findings 12 and 13, below. To be fair, we should point out that our findings are skewed in the sense that our study did not expose the many errors that are correctly caught and handled (as evidenced by the long uptime of these systems).

Nevertheless, the correctness of error handling code is particularly important given their impact. Previous studies [4, 5] show that the initial faults in distributed system failures are highly diversified (e.g., bugs, misconfigurations, node crashes, hardware faults), and in practice it is simply impossible to eliminate all of them [1]. It is therefore unavoidable that some of these faults will manifest themselves into errors, and error handling then becomes the last line of defense [3].

Trivial Mistakes in Error Handlers

Finding 12: *Thirty-five percent of the catastrophic failures are caused by trivial mistakes in error handling logic—ones that simply violate best programming practices, and that can be detected without system-specific knowledge.*

Figure 2 breaks down the trivial mistakes into three categories: (1) the error handler ignores explicit errors; (2) the error handler over-catches an exception and aborts the system; and (3) the error handler contains “TODO” or “FIXME” comments.

Twenty-five percent of the catastrophic failures were caused by ignoring explicit errors. (An error handler that only logs the error is also considered to be ignoring the error.) For systems written in Java, the exceptions were all explicitly thrown, whereas in Redis they were system call error returns.

Another 8% of the catastrophic failures were caused by developers prematurely aborting the entire cluster on a non-fatal exception. While in principle one would need system-specific knowledge to determine when to bring down the entire cluster, the aborts we observed were all within *exception over-catches*, where a higher level exception is used to catch multiple different lower-level exceptions. Figure 3 shows such an example. The `exit()` was intended only for `IncorrectVersionException`. However, the developers catch a high-level exception: `Throwable`. Consequently, when a glitch in the namenode caused `registerDatanode()` to throw `RemoteException`, it was over-caught by `Throwable` and brought down every datanode. The fix is to handle `RemoteException` explicitly.

```
try {
    namenode.registerDatanode();
+ } catch (RemoteException e) {
+   // retry.
} catch (Throwable t) {
    System.exit(-1);
}
```

RemoteException is thrown due to glitch in namenode

Only intended for IncorrectVersionException

Figure 3: An entire HDFS cluster brought down by an over-catch

User: MapReduce jobs hang when a rare Resource Manager restart occurs. I have to ssh to every one of our 4000 nodes in a cluster and try to kill all the running Application Managers.

```
Patch:
catch (IOException e) {
- // TODO
  LOG("Error event from RM: shutting down..");
+ // This can happen if RM has been restarted. Must clean up.
+ eventHandler.handle(..);
}
```

Figure 4: A catastrophic failure in MapReduce where developers left a “TODO” in the error handler

Figure 4 shows an even more obvious mistake, where the developers only left a “TODO” comment in the handler in addition to a logging statement. While this error would only occur rarely, it took down a production cluster of 4,000 nodes.

System-Specific Bugs

Fifty-seven percent of catastrophic failures are caused by incorrect error handling where system-specific knowledge is required to detect the bugs (see Figure 2).

Finding 13: *In 23% of the catastrophic failures, the mistakes in error handling were system specific, but were still easily detectable. More formally, the incorrect error handling in these cases would be exposed by 100% statement coverage testing on the error handling logic.*

In other words, once the problematic basic block in the error handling code is triggered, the failure is guaranteed to be exposed. This suggests that these basic blocks were faulty and simply never tested. The failure shown in Figure 1 belongs to this category. Once a test case can deterministically trigger the catch block, the failure will manifest with 100% certainty.

Hence, a good strategy to prevent these failures is to start from existing error handling logic and try to reverse-engineer test cases that trigger them. While high statement coverage on error handling code might seem difficult to achieve, aiming for higher statement coverage in testing might still be a better strategy than a strategy of applying random fault injections. Our finding suggests that a “bottom-up” approach could be more effective: start from the error handling logic and reverse-engineer a test case to expose errors there.

The remaining 34% of catastrophic failures involve complex bugs in the error handling logic. While our study cannot provide constructive suggestions on how to identify such bugs, we found they only account for one third of the catastrophic failures.

Aspirator: A Simple Checker

In the subsection “Trivial Mistakes in Error Handlers,” we observed that some of the most catastrophic failures are caused by trivial mistakes that fall into three simple categories: (1) error handlers that are empty or only contain log printing statements;

Simple Testing Can Prevent Most Critical Failures

(2) error handlers that over-catch exceptions and abort; and (3) error handlers that contain phrases like “TODO” and “FIXME.” We built a rule-based static checker, Aspirator, capable of locating these bug patterns. We provided two implementations of Aspirator: one as a stand-alone tool that analyzes Java bytecode, and another version that can be integrated with the Java build system to catch these bugs at compile-time. The implementation details of Aspirator can be found in our OSDI paper [6].

Checking Real-World Systems

We first evaluated Aspirator on the set of catastrophic failures used in our study. If Aspirator had been used and the identified bugs fixed, 33% of the Cassandra, HBase, HDFS, and MapReduce’s catastrophic failures we studied would have been prevented. We then used Aspirator to check the latest stable versions of these four systems plus five other systems: Cloudstack, Hive, Tomcat, Spark, and ZooKeeper.

We categorized each warning generated by Aspirator into one of three categories: bug, bad practice, and false positive. Bugs are the cases where the error handling logic will clearly lead to unexpected failures. False positives are those that clearly would not lead to a failure. Bad practices are cases that the error handling logic is suspicious of, but we could not definitively infer the consequences without domain knowledge. For example, if deleting a temporary file throws an exception and is subsequently ignored, it may be inconsequential. However, it is nevertheless considered a bad practice because it may indicate a more serious problem in the file system.

Overall, Aspirator detected 121 new bugs and 379 bad practices along with 115 false positives. Aspirator found new bugs in every system we checked.

Many bugs detected by Aspirator could indeed lead to catastrophic failures. For example, all four bugs caught by the abort-in-over-catch checker could bring down the cluster on an unexpected exception similar to the one in Figure 3. They have all been fixed by the developers of the respective systems.

Some bugs can also cause the cluster to hang. Aspirator detected five bugs in HBase and Hive that have a pattern similar to the one depicted in Figure 5(a). In this example, when tableLock cannot be released, HBase only outputs an error message and continues executing, which can deadlock all servers accessing

<pre> try { tableLock.release(); } catch (IOException e) { LOG("Can't release lock", e); } hang: lock is never released!</pre>	<pre> try { journal.recoverSegments(); } catch (IOException ex) { Cannot apply the updates from Edit log, ignoring it can cause dataloss!</pre>
--	---

Figure 5: Two new bugs found by Aspirator

the table. The developers fixed this bug by immediately cleaning up the states and aborting the problematic server.

Figure 5(b) shows a bug that could lead to data loss. An IOException could be thrown when HDFS is recovering the updates from the edit log. Ignoring this exception could cause a silent data loss.

Experience

Interaction with developers: We reported 171 bugs and bad practices to the developers of the respective systems: 143 have already been confirmed or fixed by the developers, 17 were rejected, and developers never responded to the other 11 reports.

We received mixed feedback from developers. On the one hand, positive comments include: “I really want to fix issues in this line, because I really want us to use exceptions properly and never ignore them”; “No one would have looked at this hidden feature; ignoring exceptions is bad precisely for this reason”; and “Catching Throwable [i.e., exception over-catch] is bad; we should fix these.” On the other hand, we received negative comments like: “I fail to see the reason to handle every exception.”

There are a few reasons why developers may be oblivious to the handling of errors. First, some errors are ignored because they are not regarded as critical at the time, and the importance of the error handling is realized only when the system suffers a serious failure. We hope to raise developers’ awareness by showing that many of the most catastrophic failures today are caused precisely by such obliviousness to the correctness of error handling logic.

Second, developers may believe the errors would never (or only very rarely) occur. Consider the following code snippet detected by Aspirator from HBase:

```

try {
    t = new TimeRange(timestamp, timestamp+1);
} catch (IOException e) {
    // Will never happen
}
```

In this case, the developers thought the constructor could never throw an exception, so they ignored it (as per the comment in the code). We observed many empty error handlers containing similar comments in the systems we checked. We argue that errors that “can never happen” should be handled defensively to prevent them from propagating. This is because developers’ judgment could be wrong, later code evolutions may enable the error, and allowing such unexpected errors to propagate can be deadly. In the HBase example above, developers’ judgment was indeed wrong. The constructor is implemented as follows:

```

public TimeRange (long min, long max)
throws IOException {
    if (max < min)
        throw new IOException("max < min");
}
```


Simple Testing Can Prevent Most Critical Failures

where an `IOException` is thrown on an integer overflow; yet swallowing this exception could lead to a data loss. The developers later fixed this by handling the `IOException` properly.

Third, proper handling of the errors can be difficult. It is often much harder to reason about the correctness of a system's abnormal execution path than its normal execution path. The problem is further exacerbated by the reality that many of the exceptions are thrown by poorly documented third-party components. We surmise that in many cases, even the developers may not fully understand the possible causes or the potential consequences of an exception. This is evidenced by the following code snippet from Cloudstack:

```
} catch (NoTransitionException ne) {
    / Why this can happen? Ask God not me. /
}
```

We observed similar comments from empty exception handlers in other systems as well.

Finally, feature development is often prioritized over exception handler coding when release deadlines loom. We embarrassingly experienced this ourselves when we ran Aspirator on Aspirator's code: We found five empty exception handlers, all of them for the purpose of catching exceptions thrown by the underlying libraries and put there only so that the code would compile.

Good practice in Cassandra: Among the nine systems we checked, Cassandra has the lowest bug-to-handler-block ratio, indicating that Cassandra developers are careful in following good programming practices in exception handling. In particular, the vast majority of the exceptions are handled by recursively propagating them to the callers, and are handled by top level methods in the call graphs. Interestingly, among the five systems we studied, Cassandra also has the lowest rate of catastrophic failures in its randomly sampled failure set (see Table 1).

Reactions from HBase developers: Our OSDI paper prompted HBase developers to start the initiative to fix all the existing bad practices. They intend to use Aspirator as their compile-time checker [2].

Conclusions

We presented an in-depth analysis of 198 user-reported failures in five widely used, data-intensive distributed systems. We found that the error-manifestation sequences leading to the failures to be relatively complex. However, we also found that almost all of the most catastrophic failures are caused by incorrect error handling, and more than half of them are trivial mistakes or can be exposed by statement coverage testing.

Existing testing techniques will find it difficult to successfully uncover many of these error-handling bugs. They all use a “top-down” approach: start the system using generic inputs and actively

inject errors at different stages. However, the size of the input and state space makes the problem of exposing these bugs intractable.

Instead, we suggest a three-pronged approach to expose these bugs: (1) use a tool similar to the Aspirator that is capable of identifying a number of trivial bugs; (2) enforce code reviews on error-handling code, since the error-handling logic is often simply wrong; and (3) purposefully construct test cases that can reach each error-handling code block.

Our detailed analysis of the failures and the source code of Aspirator are publicly available at: <http://www.eecg.toronto.edu/failureAnalysis/>.

Acknowledgments

We greatly appreciate the anonymous OSDI reviewers, Jason Flinn, Leonid Ryzhyk, Ashvin Goel, David Lie, and Rik Farrow for their insightful feedback. We thank Dongcai Shen for help with reproducing five bugs. This research is supported by an NSERC Discovery grant, NetApp Faculty Fellowship, and Connaught New Researcher Award.

References

- [1] J. Dean, “Underneath the Covers at Google: Current Systems and Future Directions,” in *Google I/O*, 2008.
- [2] HBase-12187: review in source the paper “Simple Testing Can Prevent Most Critical Failures”: <https://issues.apache.org/jira/browse/HBASE-12187>.
- [3] P.D. Marinescu and G. Candea, “Efficient Testing of Recovery Code Using Fault Injection,” *ACM Transaction on Computer Systems*, vol. 29, no. 4, Dec. 2011.
- [4] D. Oppenheimer, A. Ganapathi, and D.A. Patterson, “Why Do Internet Services Fail, and What Can Be Done About It?” in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems, USITS '03*, 2003, pp. 1–15.
- [5] A. Rabkin and R. Katz, “How Hadoop Clusters Break,” *Software, IEEE*, vol. 30, no. 4, 2013, pp. 88–94.
- [6] D. Yuan, Y. Luo, X. Zhuang, G.R. Rodrigues, X. Zhao, Y. Zhang, P.U. Jain, and M. Stumm, “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI '14*, 2014, pp. 249–265.

Nail

A Practical Tool for Parsing and Generating Data Formats

JULIAN BANGERT AND NICKOLAI ZELDOVICH



Julian Bangert is a second year PhD student working on computer security at MIT. When he is not building parsers for complicated

formats, he is interested in building exploit mitigation techniques, side-channel resistant cryptography, and finding Turing-complete weird machines in unexpected places, such as your processor's virtual memory system. julian@csail.mit.edu.



Nikolai Zeldovich is an associate professor at MIT's Department of Electrical Engineering and Computer Science and a member

of the Computer Science and Artificial Intelligence Laboratory. His research interests are in building practical secure systems, from operating systems and hardware to programming languages and security analysis tools. He received his PhD from Stanford University in 2008, where he developed HiStar, an operating system designed to minimize the amount of trusted code by controlling information flow. In 2005, he co-founded MokaFive, a company focused on improving desktop management and mobility using x86 virtualization. Professor Zeldovich received a Sloan Fellowship (2010), an NSF CAREER Award (2011), the MIT EECS Spira Teaching Award (2013), and the MIT Edgerton Faculty Achievement Award (2014). nickolai@csail.mit.edu

I am ZIP, file of files. Parse me, ye mighty and drop a shell.

—Edward Shelley on the Android Master Key

Binary file formats and network protocols are hard to parse safely: The libpng image decompression library had 24 remotely exploitable vulnerabilities from 2007 to 2013. According to CVE details, Adobe's PDF and Flash viewers have been notoriously plagued by input processing vulnerabilities, and even the zlib compression library had input processing vulnerabilities in the past. Most of these attacks involve memory corruption—therefore, it is easy to assume that solving memory corruption will end all our woes when handling untrusted inputs.

However, as memory-safe languages and exploit mitigation tricks are becoming more prevalent, attackers are moving to a new class of attack—parser differentials. Many applications use handwritten input processing code, which is often mixed with the rest of the application—e.g., by passing a pointer to the raw input through the application. This (anti-) pattern makes it impossible to figure out whether two implementations of the same format or protocol are identical, and input handling code can't be easily reused between applications. As a result, different applications often disagree in the corner cases of a protocol, which can have fatal security consequences. For example, Android has two parsers for ZIP archives involved in securely installing applications. First, a Java program checks the signatures of files contained within an app archive and then another tool extracts them to the file system. Because the two ZIP parsers disagree in multiple places, attackers can modify a valid file so that the verifier will see the original contents, but attacker-controlled files will be extracted to the file system, bypassing Android's code signing. Similar issues showed up on iOS [3] and SSL [4].

Instead of attempting to parse inputs by hand and failing, a promising approach is to specify a precise grammar for the input data format and automatically generate parsing code from that with tools like yacc. As long as the parser generator is bug-free, the application will be safe from many input processing vulnerabilities. Grammars can also be reused between applications, further reducing effort and eliminating inconsistencies.

This approach is typical in compiler design and in other applications handling text-based inputs, but not common for binary inputs. The Hammer framework [5] and data description languages such as PADS [2] have been developing generated parsers for binary protocols.

However, if you wanted to use existing tools to parse PDF or ZIP, you would soon find that they cannot handle the complicated—and therefore most error-prone—aspects of such formats, so you'd still have to handwrite the riskiest bits of code. For example, existing parser generators cannot conveniently represent size or offset fields, and more complex features, such as data compression or checksums, cannot be expressed at all.

Nail: A Practical Tool for Parsing and Generating Data Formats

```

1  dnspacket =
2  {
3  id uint16
4  qr uint1
5  opcode uint4
6  aa uint1
7  tc uint1
8  rd uint1
9  ra uint1
10 uint3 = 0
11 rcode uint4
12 @qc uint16
13 @ac uint16
14 @ns uint16
15 @ar uint16
16 questions n_of @qc question
17 responses n_of @ac answer
18 authority n_of @ns answer
19 additional n_of @ar answer
20 }
21 question = {
22 labels compressed_labels
23 qtype uint16 | 1..16
24 qclass uint16 | [1,255]
25 }
26 answer = {
27 labels compressed_labels
28 rtype uint16 | 1..16
29 class uint16 | [1]
30 ttl uint32
31 @rlength uint16
32 rdata n_of @rlength uint8
33 }
34 compressed_labels = {
35 $decompressed transform dnscompress ($current)
36 labels apply $decompressed labels
37 }
38 label = { @length uint8 | 1..64
39           label n_of @length uint8 }
40 labels = <many label; uint8 = 0>

```

Figure 1: Nail grammar for DNS packets, used by our prototype DNS server

Furthermore, some parser generators are cumbersome to use when parsing binary data for several reasons. First, many parser generators don't produce convenient data structures, but call semantic actions that you have to write to build up a data structure your program can use. Therefore, you must describe the format up to three times—in the grammar, the data structure, and the semantic actions. Second, most parser generators only address parsing inputs, so you have to manually construct outputs. Some parser generators, such as Boost.Spirit, allow

generating output but require you to write another set of semantic actions.

We address these challenges with Nail, a new parser generator for binary formats. First, Nail grammars describe not only a format, but also a data type to represent it within the program. Therefore, you don't have to write semantic actions and type declarations, and you can no longer combine syntactic validation and semantic processing. Second, Nail will also generate output from this data type without requiring you to write more risky code or giving you a chance to introduce ambiguity.

Third, Nail introduces two abstractions, *dependent fields* and *transformations*, to elegantly handle problematic structures, such as offset fields or checksums. Dependent fields capture fields in a protocol whose value depends in some way on the value or layout of other parts of the format; for example, offset or length fields, which specify the position or length of another data structure, fall into this category. Transformations allow you to write plugins, allowing your programs to handle complicated structures, while keeping Nail itself small, yet flexible.

In the rest of this article, we will show some tricky features of real-world formats and how to handle them with Nail.

Design by Example

In this section, we will explain how to handle basic data formats in Nail, how to handle redundancies in the format with dependent fields, and how Nail parsers can be extended with transformations.

As a motivating example, we will parse DNS packets, as defined in RFC 1035. Each DNS packet consists of a header, a set of question records, and a set of answer records. Domain names in both queries and answers are encoded as a sequence of labels, terminated by a zero byte. Labels are Pascal-style strings, consisting of a length field followed by that many bytes comprising the label.

Basic Data Formats

Let's step through a simplified Nail grammar for DNS packets, shown in Figure 1. For this grammar, Nail produces the type declarations shown in Figure 2 and the parser and generator functions shown in Figure 3. Nail grammars are reusable between applications, and we will use this grammar to implement both a DNS server and a client, which previously would have had two separate handwritten parsers, leading to bugs such as the Android Master Key.

A Nail grammar file consists of rule definitions—for example, lines 1–20 of Figure 1 assign a name (`dnspacket`) to a grammar production (lines 2–20). If you are not familiar with other parsers, you can imagine rules as C type declarations on steroids (although our syntax is inspired by Go).

Nail: A Practical Tool for Parsing and Generating Data Formats

```

struct dnspacket {
    uint16_t id;
    uint8_t qr;
    /* ... */
    struct {
        struct question *elem;
        size_t count;
    } questions;
};

```

Figure 2: Portions of the C data structures defined by Nail for the DNS grammar shown in Figure 1

```

struct dnspacket *parse_dnspacket(NailArena *arena,
    const uint8_t *data,
    size_t size);
int gen_dnspacket(NailArena *tmp_arena,
    NailStream *out,
    struct dnspacket *val);

```

Figure 3: The API functions generated by Nail for parsing inputs and generating outputs for the DNS grammar shown in Figure 1

Just as C supports various constructs to build up types, such as structures and unions from pointers and elemental types, Nail supports various *combinators* to represent features of a file or protocol. We will present the features we used in implementing DNS. A more complete reference can be found in [4], with a detailed rationale in [1].

Integers and Constraints. Because Nail is designed to cope with binary formats, it handles not only common integer types (e.g., `uint16`) but bit fields of any length, such as `uint1`. These integers are exposed to the programmer as an appropriately sized machine integer (e.g., `uint8_t`). Nail also supports constraints on integer values, limiting the values to either a range (line 23, `1..16`), which can optionally be half open or a fixed set (line 24, `[1,255]`). Both types of constraint can be combined, e.g., `[1..16,255]`. Constant values are also supported—e.g., line 10: `uint3=0` represents three reserved bits that must be 0. Because constant values carry no information, they are not represented in the data type.

Structures. The body of the `dnspacket` rule is a structure, which contains any number of fields enclosed between curly braces. Each field in the structure is parsed in sequence and represented as a structure to the programmer. Contrary to other programming languages, Nail does not have a special keyword for structs. We also reverse the usual structure-field syntax: `id uint1` is a field called `id` with type `uint1`. Often, Nail grammars have structures with just one non-constant field—for example, when parsing a fixed header. Nail supports this with an alternative form of structures, using angle brackets, that contains one

unnamed, non-constant field, which is represented directly in the datatype, without introducing another layer of indirection, as shown on line 40.

Arrays. Nail supports various forms of arrays. Line 40 shows how to parse a domain in a DNS packet with `many`, which keeps repeating the `label` rule until it fails. In the next section, we will explain how to handle count fields, and our full paper describes how to handle various array representations (such as delimiters or non-empty arrays).

Redundant Data

Data formats often contain values that are determined by other values or the layout of information, such as checksums, duplicated information, or offset and length fields. Exposing such values risks inconsistencies that could trick the program into unsafe behavior. Therefore, we represent such values using *dependent fields* and handle them transparently during parsing and generation without exposing them to the application. Dependent fields are handled like other fields when parsing input but are only stored temporarily instead of in the data type. Their value can be referenced by other parsers until it goes out of scope. When generating output, Nail inserts the correct value.

In DNS packets, the packet header contains count fields (`qc`, `ac`, `ns`, and `ar`), which contain the number of questions and answers that follow the header and which we represent by dependent fields (lines 12–15). Dependent fields are defined within a structure like normal fields, but their name starts with an `@` symbol. A dependent field is in scope and can be referred to by the definition of all subsequent fields in the same structure. Dependent fields can be passed to rule invocations as parameters.

Nail allows handling count fields with `n_of`, which parses an exact number of repetitions of a rule. Lines 16–19 in Figure 1 show how to use `n_of` to parse the question and answer records in a DNS packet. Other dependencies, such as offset fields or checksums, are not handled directly by combinators but through transformations, as we describe next.

Input Streams and Transformations

So far, we have described a parser that consumes input a byte at a time from beginning to end. However, real-world formats often require nonlinear parsing. Offset fields require a parser to move to a different position in the input, possibly backwards. Size fields require the parser to stop processing before the end of input has been reached. Other cases, such as compressed data and checksums, require more complicated processing on parts of the input before it can be handled.

For a parser to be useful, it needs to support all these ways of structuring a format. This is why data description languages like PADS [2] contain not just a kitchen sink, but a kitchen store

Nail: A Practical Tool for Parsing and Generating Data Formats

Nail Grammar	External Format	Internal Data Type in C
<code>uint4</code>	4-bit unsigned integer	<code>uint8_t</code>
<code>int32 [1,5..255,512]</code>	Signed 32-bit integer $x \in \{1,5..255,512\}$	<code>int32_t</code>
<code>uint8 = 0</code>	8-bit constant with value 0	<code>/* empty */</code>
<code>optional int8 16..</code>	8-bit integer ≥ 16 or nothing	<code>int8_t *</code>
<code>many int8 ![0]</code>	A NULL-terminated string	<pre>struct { size_t N_count; int_t *elem; };</pre>
<pre>{ hours uint8 minutes uint8 }</pre>	Structure with two fields	<pre>struct { uint8_t hours; uint8_t minutes; };</pre>
<code><int8=""; p; int8=""></code>	A value described by parser <i>p</i> , in quotes	The data type of <i>p</i>
<pre>choose { A = uint8 1..8 B = uint16 256.. }</pre>	Either an 8-bit integer between 1 and 8, or a 16-bit integer larger than 256	<pre>struct { enum {A, B} N_type; union { uint8_t a; uint16_t b; }; };</pre>
<pre>@valuelen uint16 value n_of @valuelen uint8</pre>	A 16-bit length field, followed by that many bytes	<pre>struct { size_t N_count; uint8_t *elem; };</pre>
<pre>\$data transform deflate(\$current @method)</pre>	Applies programmer-specified function to create new stream	<code>/* empty */</code>
<code>apply \$stream p</code>	Apply parser <i>p</i> to stream <i>\$stream</i>	The data type of <i>p</i>
<code>foo = p</code>	Define rule <i>foo</i> as parser <i>p</i>	<code>typedef /* type of <i>p</i> */ foo;</code>
<code>* p</code>	Apply parser <i>p</i>	Pointer to the data type of <i>p</i>

Figure 4: Syntax of Nail parser declarations and the formats and data types they describe

full of features, and a language that can handle all possible formats will be a general purpose programming language. Instead, we keep Nail itself small and introduce an interface that allows complicated format structures to be handled by plugin *transformations* in a general purpose language. Of course, we ship Nail with a handy library of common transformations to handle common format features, such as offsets, sizes, and checksums.

These *transformations* consume and produce streams—sequences of bytes—which can be further passed to other transformations and eventually parsed by a Nail rule. Transformations can also access values in dependent fields. Streams can be subsets of other streams: for example, the substream starting at an offset given in a dependent field to handle pointer fields, or computed at runtime, such as by decompressing another stream with zlib.

Nail: A Practical Tool for Parsing and Generating Data Formats

Transformations are two arbitrary functions called during parsing and output generation. The parsing function consumes any number of streams and dependent field values, and produces any number of temporary streams. This function may reposition and read from the input streams and read the values of dependent fields, but not change their contents and values. The generating function has to be an inverse of the parsing function, consuming streams and producing dependent field values and other streams.

As a concrete example, we will show a grammar for ProtoZIP, a very simple archive format inspired by ZIP in Figure 5. ProtoZIP consists of a variable-length end-of-file directory, which is a magic number followed by an array of filenames and pointers to compressed files. A grammar for the real ZIP format, which has more layers of indirection, is presented in the full paper.

In Figure 5, the grammar first calls the `zipdir` transform on line 2, which finds the magic number and splits the file into two streams, one containing the compressed files, the other the directory. Streams are referred to with `$identifiers`, similar to dependent fields. A C prototype of the `zipdir` transform is shown in Figure 6.

When parsing input, this will call `zipdir_parse`, which takes `$current`—an implicit identifier always referring to the stream currently being handled—and returns `$files` and `$header`. When generating output, this will call `zipdir_generate`, which appends `$files` and `$header` to `$current`.

Line 3 of Figure 5 then applies the `dir` rule to the `$header` stream, passing it the `$files` stream. Within `dir`, `$current` is now `$header` and input is parsed from and output generated to that stream. The `dir` rule in turn describes the structure of the directory—a magic number and a count field, followed by that many file descriptors. Each file descriptor is then parsed with two transformations: the standard-library `slice`, which describes an offset and a size within another stream, and the custom `zlib`, which compresses a stream using `zlib`. Finally, we apply a trivial grammar (line 14) to the contents.

In a more complicated example, such as an Office document, we could now specify grammars for each entry within an archive.

Transformations need to be carefully written, because they can violate Nail's safety properties and introduce bugs. However, as we will show below (see Applications), Nail transformations are much shorter than handwritten parsers, and many formats can be represented with just the transformations in Nail's standard library. For example, our Zip transformations are 78 lines of code, compared to 1600 lines of code for a handwritten parser. Additionally, Nail provides convenient and safe interfaces for allocating memory and accessing streams that address the most common occurrences of buffer overflow vulnerabilities.

```
1 protozip = {
2   $files, $header transform zipdir($current)
3   contents apply $header dir($files)
4 }
5 dir($files) = {
6   uint32 = 0x00034b50
7   @count uint32
8   files n_of @count {
9     @off uint32
10    @size uint32
11    filename many (uint8 | ![0])
12    $compr transform slice_u32($files @off @size)
13    $decomp transform zlib($compr)
14    contents apply $decomp (many uint8)
15  }
16 }
```

Figure 5: Nail grammar for ZIP files. Various fields have been cut for brevity.

```
int zip_end_of_directory_parse(
    NailArena *tmp, NailStream *out_files,
    NailStream *out_dir, NailStream *in_current);
int zip_end_of_directory_generate(
    NailArena *tmp, NailStream *in_files,
    NailStream *in_dir, NailStream *out_current);
```

Figure 6: Signatures of stream transform functions for handling the end-to-beginning structure of ProtoZIP files

Using Nail

Real-World Formats

We used Nail to implement grammars for seven protocols with a range of challenging features. Figure 7 summarizes our results. Despite the challenging aspects of these protocols, Nail is able to capture them by relying on its novel features: dependent fields, streams, and transforms. In contrast, state-of-the-art parser generators would be unable to fully handle five out of the seven data formats.

DNS. Previously, we used a grammar for DNS packets shown in Figure 1 to show how to write Nail grammars. This example grammar corresponds almost directly to the diagrams in RFC 1035, which defines DNS. Nail's dependent fields handle DNS's count fields, and transformations represent label compression. At best, both of these features are awkward to handle with existing tools.

ZIP. An especially tricky data format is the ZIP compressed archive format, as specified by PKWARE. At the end of each ZIP file is an *end-of-directory header*. This header contains a variable-length comment, so it has to be located by scanning backwards from the end of the file until a magic number and a valid length field are found. Many ZIP implementations disagree

Nail: A Practical Tool for Parsing and Generating Data Formats

Protocol	LoC	Challenging Features
DNS packets	48+64	Label compression, count fields
ZIP archives	92+78	Checksums, offsets, variable length trailer, compression
Ethernet	16+0	—
ARP	10+0	—
IP	25+0	Total length field, options
UDP	7+0	Checksum, length field
ICMP	5+0	Checksum

Figure 7: Protocols, sizes of their Nail grammars, and challenging aspects of the protocol that cannot be expressed in existing grammar languages. A + symbol counts lines of Nail grammar code (before the +) and lines of C code for protocol-specific transforms (after the +).

on the exact semantics of this, such as when the comment contains the magic number [6]. This header contains the offset and the size of the *ZIP directory*, which is an array of *directory entry headers*, one for every file in the archive. Each entry stores file metadata in addition to the offset of a *local file header*. The local file header duplicates most information from the directory entry header and is followed immediately by the compressed archive entry. Duplicating information made sense when ZIP files were stored on floppy disks with slow seek times and high fault rates, but nowadays it leads to parsers being confused, such as in the recent Android Master Key bug.

Nail captures these redundancies with dependent fields, eliminating the ambiguities. It also decompresses archive contents transparently with transformations, which allows parsing the contents of an archive file—allowing formats based on ZIP, such as Microsoft Office documents, to be handled with one grammar.

Applications

We implemented two applications—a DNS server and an unzip program—based on the above grammars, and will compare the effort involved and the resulting security to similar applications with handwritten parsers and with other parser generators. We will use lines of code as a proxy for programmer effort. To evaluate security, we will argue how our design avoids classes of vulnerabilities and fuzz-test one of our applications.

DNS. Our DNS server parses a zone file, listens to incoming DNS requests, parses them, and generates appropriate responses. The DNS server is implemented in 183 lines of C, together with 48 lines of Nail grammar and 64 lines of C code implementing stream transforms for DNS label compression. In comparison, Hammer [5] ships with a toy DNS server that responds to any valid DNS query with a CNAME record to the domain “spargelze.it”. Their server consists of 683 lines of C, mostly custom validators, semantic actions, and data structure

Application	LoC w/ Nail	LoC w/o Nail
DNS server	295	683 (Hammer parser)
unzip	220	1,600 (Info-Zip)

Figure 8: Comparison of code size for two applications written in Nail, and a comparable existing implementation without Nail

definitions, with 52 lines of code defining the grammar with Hammer’s combinators. Their DNS server does not implement label compression, zone files, etc.

To evaluate whether Nail-based parsers are compatible with good performance, we compare the performance of our DNS server to that of ISC BIND 9 release 9.9.5, a mature and widely used DNS server. We simulate a load resembling that of an authoritative name server, generating a random zone file and a random sequence of queries, with 10% non-existent domains. We repeated this sequence of queries for one minute against both DNS servers. We found that our DNS server is approximately three times faster than BIND. Although BIND is a more sophisticated DNS server and implements many features that are not present in our Nail-based DNS server and that allow it to be used in more complicated configurations, we believe our results demonstrate that Nail’s parsers are not a barrier to achieving good performance.

ZIP. We implemented a ZIP file extractor in 50 lines of C code, together with 92 lines of Nail grammar and 78 lines of C code implementing two stream transforms (one for the DEFLATE compression algorithm with the help of the zlib library, and one for finding the end-of-directory header). The unzip utility contains a file `extract.c`, which parses ZIP metadata and calls various decompression routines in other files. This file measures over 1,600 lines of C, which suggests that Nail is highly effective at reducing manual input parsing code, even for the complex ZIP file format.

In our full paper [1], we present a study of 15 ZIP parsing bugs. Eleven of these vulnerabilities involved memory corruption during input handling, which Nail’s generated code is immune to by design. We also fuzz-tested our DNS server. More interestingly, Nail also protects against parsing inconsistency vulnerabilities like the four others we studied. Nail grammars explicitly encode duplicated information such as the redundant length fields in ZIP that caused a vulnerability in the Python ZIP library. The other three vulnerabilities exist because multiple implementations of the same protocol disagree on some inputs. Handwritten protocol parsers are not very reusable, as they build application-specific data structures and are tightly coupled to the rest of the code. Nail grammars, however, can be reused between applications, avoiding protocol misunderstandings.

Nail: A Practical Tool for Parsing and Generating Data Formats

Conclusion

We presented the design and implementation of *Nail*, a tool for parsing and generating complex data formats based on a precise grammar. This helps programmers avoid memory corruption and inconsistency vulnerabilities while reducing effort in parsing and generating real-world protocols and file formats. Nail captures complex data formats by introducing *dependent fields*, *streams*, and *transforms*. Using these techniques, Nail is able to support DNS packet and ZIP file formats, and enables applications to handle these data formats in many fewer lines of code.

Nail and all of the applications and grammars developed in this paper are released as open-source software, available at <https://github.com/jbangert/nail>. A more detailed discussion of our design and our results is available in [1].

Acknowledgments

We thank M. Frans Kaashoek, the OSDI reviewers, and K. Park for their feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

- [1] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 615–628, Broomfield, CO, Oct. 2014, USENIX Association: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert>.
- [2] K. Fisher and R. Gruber, "PADS: A Domain-Specific Language for Processing Ad Hoc Data," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 295–304, Chicago, IL, June 2005.
- [3] G. Hotz, *evasi0n 7 writeup*, 2013: <http://geohot.com/e7writeup.html>.
- [4] D. Kaminsky, M.L. Patterson, and L. Sassaman, "PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure," in *Proceedings of the 2010 Conference on Financial Cryptography and Data Security*, pp. 289–303, Jan. 2010.
- [5] M. Patterson and D. Hirsch, *Hammer parser generator*, March 2014: <https://github.com/UpstandingHackers/hammer>.
- [6] J. Wolf, "Stupid ZIP file tricks!" in *BerlinSides 0x7DD*, 2013.



Publish and Present Your Work at USENIX Conferences

The program committees of the following conferences are seeking submissions. CiteSeer ranks the USENIX Conference Proceedings among the top ten highest-impact publication venues for computer science.

Get more details about these Calls at www.usenix.org/cfp.

USENIX Security '15: 24th USENIX Security Symposium

August 12–14, 2015, Washington, D.C.

Paper titles and abstracts, as well as invited talk and panel proposals due: February 16, 2015

The USENIX Security Symposium brings together researchers, practitioners, system administrators, system programmers, and others interested in the latest advances in the security and privacy of computer systems and networks. All researchers are encouraged to submit papers covering novel and scientifically significant practical works in computer security. USENIX Security '15 papers may be submitted for consideration for the Internet Defense Prize. The Symposium also seeks Poster and Work-in-Progress (WIP) submissions as well as proposals for invited talks and panel discussions.

HotCloud '15: 7th USENIX Workshop on Hot Topics in Cloud Computing

July 6–7, 2015, Santa Clara, CA

Submissions due: March 10, 2015

HotCloud brings together researchers and practitioners from academia and industry working on cloud computing technologies to share their perspectives, report on recent developments, discuss research in progress, and identify new/emerging "hot" trends in this important area. While cloud computing has gained traction over the past few years, many challenges remain in the design, implementation, and deployment of cloud computing.

HotStorage '15: 7th USENIX Workshop on Hot Topics in Storage and File Systems

July 6–7, 2015, Santa Clara, CA

Submissions due: March 17, 2015

The purpose of the HotStorage workshop is to provide a forum for the cutting edge in storage research, where researchers can exchange ideas and engage in discussions with their colleagues. The workshop seeks submissions that explore long term challenges and opportunities for the storage research community. Submissions should propose new research directions, advocate non-traditional approaches, or report on noteworthy actual experience in an emerging area. We particularly value submissions that effectively advocate fresh, unorthodox, unexpected, controversial, or counterintuitive ideas for advancing the state of the art.

LISA15

November 8–13, 2015, Washington, D.C.

Submissions due: April 17, 2015

The LISA conference is the premier conference for IT operations, where systems engineers, operations professionals, and academic researchers share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world. LISA invites submissions of proposals from industry leaders for talks, mini-tutorials, tutorials, panels and workshops. LISA is also interested in research related to the fields of system administration and engineering. We welcome submissions for both research papers and posters.

CSET '15: 8th Workshop on Cyber Security Experimentation and Test

August 10, 2015, Washington, D.C.

Submissions due: April 23, 2015

CSET invites submissions on the science of cyber security evaluation as well as experimentation, measurement, metrics, data, and simulations, as those subjects relate to computer and network security and privacy. The "science" of cyber security poses significant challenges: very little data are available for research use, and little is understood about what good data would look like if it were obtained. Experiments must recreate relevant, realistic features—including human behavior—in order to be meaningful, yet identifying those features and modeling them is hard. Repeatability and measurement accuracy are essential in any scientific experiment, yet hard to achieve in practice. Cyber security experiments carry significant legal and ethical risks if not properly contained and controlled, yet often require some degree of interaction with the larger world in order to be useful. Meeting these challenges requires transformational advances, including understanding the relationship between scientific method and cyber security evaluation, advancing capabilities of underlying experimental infrastructure, and improving data usability.

3GSE '15: 2015 USENIX Summit on Gaming, Games and Gamification in Security Education

August 10, 2015, Washington, D.C.

Submissions due: May 5, 2015

3GSE '15 is designed to bring together educators and game designers working in the growing field of digital games, non-digital games, pervasive games, gamification, contests, and competitions for computer security education. The summit will attempt to represent, through invited talks, paper presentations, panels, and tutorials, a variety of approaches and issues related to using games for security education.

FOCI '15: 5th USENIX Workshop on Free and Open Communications on the Internet

August 10, 2015, Washington, D.C.

Submissions due: May 12, 2015

Internet communications drive political and social change around the world. Governments and other actors seek to control, monitor, manipulate and block Internet communications for a variety of reasons, ranging from extending copyright law to suppressing free speech and assembly. Methods for controlling what content people post and view online are also multifarious. Whether it's traffic throttling by ISPs or man-in-the-middle attacks by countries seeking to identify those organizing protests, threats to free and open communications on the Internet raise a wide range of research challenges.

Capacity Planning

DAVID HIXSON AND KAVITA GULIANI



David Hixson is a technical project manager in Site Reliability Engineering at Google, where he has been for eight years. He currently spends

his time predicting how social products at Google will grow and trying to make the reality better than the plan. He previously worked as a system administrator on high availability systems and has an MBA from Arizona State. <https://plus.google.com/+DavidHixson>, dhixson@google.com



Kavita Guliani is a technical writer for Technical Infrastructure and Site Reliability Engineering at Google, Mountain View.

Before working at Google, Kavita worked for companies like Symantec, Cisco, and Lam Research Corporation. She holds a degree in English from Delhi University and studied technical writing at San Jose State University. kguliani@google.com

Capacity planning can be a torturous exercise in spreadsheets and meetings that drains the life out of junior engineers, provides little value to the company, leads to ongoing recriminations for everyone involved, and results in little planning or capacity.

Alternatively, it can be used to hone the understanding of the core services being offered, to work across the company to understand risks, and to make thoughtful choices for the business.

Let's focus on this second approach and talk about how three different parts of the company can play an active role in capacity planning. The more the players understand their part in the greater scheme, the better they can communicate and make tradeoffs that benefit the company.

Figure 1 represents three different perspectives. They may all be the same person thinking about the problem differently or they might be three vast organizations that rarely manage to get people into the same room. Even within a company, some products might need to be evaluated at different levels based upon their potential impact. The thought process is much more important than the job title associated with it.

What Is Capacity?

Before we get too far, let us define what capacity is. Very simply, capacity consists of the resources required to run your service or services in the context you have chosen to run them. The very core of this may be subject to debate or change, but the key things to predict are the resources you are constrained by. Depending on your scale or architecture, this could be gigs of RAM on a machine in your bedroom, cloud VMs, physical computers at your colo, bandwidth on a CDN, or megawatts of power.

As you increase in scale and complexity, you probably start to experience pressure in several dimensions, perhaps network capacity as well as storage or compute. And you may have different scaling limitations based upon your choices, either in terms of flexibility or timing around growth.

Engineering

Traditionally, the engineering organization would own the technical complexity and have the best understanding of how the system works right now, what choices were made to get here, and what things might be done in the future.

Depending on the organization, this might require cooperation across different teams, but the starting premise is that someone knows how things work and the resources required for the system to function today. This is far from trivial, but definitely a starting point when planning for the future.

Bottom-Up Capacity

The first step is to map out current system capacity. What resources does it use in order to get the work done? Identify all the large parts of the system: things that are material to your capacity-planning needs. Materiality will depend upon your organization, but there is a huge

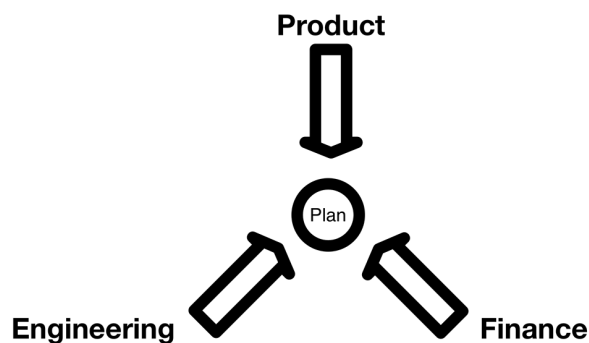


Figure 1: Three forces that impact capacity planning

value in simplicity, so if you can reduce the number of things you need to worry about, it will make everyone's life easier.

So, here we assume that you know how much you are using today in the resource dimensions that make sense for your service.

PRIMARY DRIVERS

The next step is to figure out why you are using the resources you are using today and what would make those numbers change. Metrics like “gigs of data uploaded by users today” are likely to impact your storage and bandwidth directly. Web queries per second (QPS) are likely to impact compute and, possibly, networking. Finding the fewest number of drivers (QPS, gigs uploaded, etc.) that capture the vast majority of the demand on your system is a challenge that should be reasonably thought-provoking.

If you have different types of queries or page views, you may want to look at a “costed” metric as a way to normalize the work and make it easier to understand. For example, a database read may be extremely cheap, but a write may be very costly in terms of CPU consumed and disk I/O required. So if you measure your database just in terms of QPS, you may make poor assumptions about scalability. If you can assign different “costs” to different actions, you can normalize them and make them better predictors of your ability to scale. Ideally, this is automated and constantly recalculated. However, even a gross estimate is useful. You may also realize that something like “bandwidth” is a better estimator than QPS and want to use that instead. Understanding these “costed” metrics can provide value in explaining the system as well as predicting future usage.

Popular metrics for product reporting include 30-day active or seven-day active users, which almost definitely do not determine the resources required to support your product. Similarly, you need to aggregate the data finely enough to be able to provision your system for peaks in demand. A queries-per-day metric is unlikely to be helpful here. Instead, you want QPS over a short

interval (~minutes, hopefully) so that you can identify the peaks and be prepared to survive them.

Once you uncover the likely drivers for growth within the system, start collecting data. You will need to understand how these change over time as well as how the system load changes. The combination of these is the key to your bottom-up plan.

THEORETICAL MINIMUM CAPACITY

Finding correlations (and, hopefully, causality) between your identified growth metrics and observed capacity is a bit of a holy grail. For a complex system, it can be surprisingly difficult to get the two of them to line up nicely. You may never get perfection, but a thought model around “theoretical minimum blow-up” is a good way to start looking at it, and it has a nice side effect that we will get to in a minute. The “blow-up” that we are looking for is the inflation of either data or work that is inherent in the design.

Step back from your measurements and drivers and think about what your system is really trying to accomplish. The simplest example might be backing up bytes for users. If a user gives you a byte of data and you promise to give it back upon request, you've got a really clear understanding of the product. At no point will you need to store less than that byte (probably).

So how many bytes of disk do you use to store that byte?

1 for the original byte * 1.2 for RAID5 * 1.3 for “overhead” (file system, metadata, caching, backups, operational slack), then * 2 the whole mess for our second site. So we used a total of 3.12 bytes to store the byte that user gave us.

The same kind of thing can be done for the CPU required to update your database. How many replicas, stored procedures, and other things have to happen? You almost invariably do a lot of work many times over in order to make a write. Reads probably have a different set of factors.

With this kind of model in mind, you can go down two very interesting paths:

- ◆ You might tie together your capacity drivers and observed growth in a more natural way. Using the disk example, it might be better to explain why you grow disk capacity 3x as fast as users are uploading bytes.
- ◆ You could identify a bunch of questions around how you engineered your product. If you now seek to drive all of your multiples to 1, you will go out of business shortly because it isn't about optimizing without thinking. Instead, you need to evaluate each of them and see whether they are doing what you intend. Is that 6-disk RAID configuration the one you wanted for availability? Do you need that second site, or should you have more than two? Should you be doubling your investment in caching, or is it no longer providing the value you expected?

Capacity Planning

You can also use this blow-up model to look at the engineering changes that you have planned for the future, and account for them more clearly. A wise engineer will also check in regularly to make sure that these blow-up factors and assumptions remain accurate. It may help you spot when your system is drifting away from how it was intended to operate.

Although it is possible to make this kind of model as complicated as you have time to work on it, the key thing is to pull out the large drivers of relevant capacity and to highlight the engineering tradeoffs. You get the largest benefit if you ignore all the little things, letting everyone involved focus their attention on the choices that matter.

PAST PREDICTS FUTURE

The best starting point for predicting the future is observing the past. It is far from perfect, but the alternatives all involve significantly more made-up numbers.

Using the theoretical minimum blow-up factors, extract out your historical growth, then project it into the future. Is it a curve, is it a line, is it some complicated pattern too deep for the human mind to comprehend? Maybe. But in the vast majority of cases, you can assume that it is a line that extends out since the last time you made a significant product change.

Growth is frequently broken into two categories: organic growth and inorganic growth.

Organic growth comes from the natural adoption and usage of your product by customers. It may change over time, but it should change comparatively gradually and not as a step function. Examples of organic growth in an image-serving system might include uploading more photos, resulting in more bytes that need to be stored, and having more people view the photos resulting in more network load and larger serving capacity.

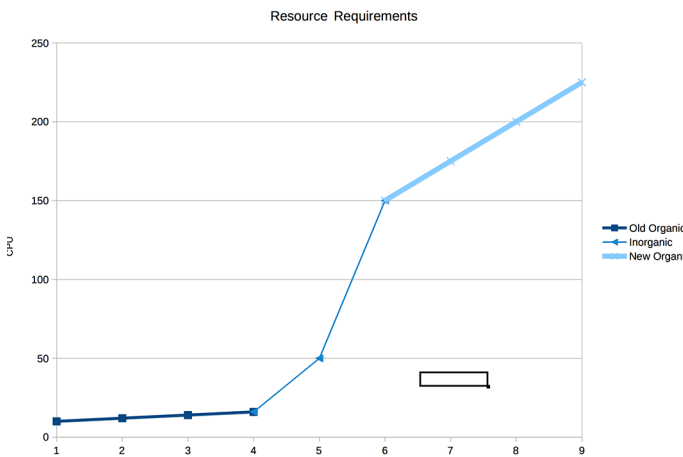


Figure 2: Examples of organic and inorganic growth over time

Inorganic growth refers to those step changes, likely the result of a feature launch, a marketing campaign, or business-driven change to how your product is being used (e.g., acquired another company and redirected traffic, bundled with another product, changed pricing, etc.)

Then you can layer on expectations about future inorganic changes. Perhaps it is the next advertising campaign or an upcoming change in the business strategy, or just a change of the background color for your app to a really soothing shade of teal. Estimate the impact and layer that into your plans, but keep them itemized because that gives you room to learn as you repeat the process and a place to begin discussions with the product and finance representatives. You should also consider how quickly you start to treat past launches as part of the organic line. As you start to observe the actual behavior, you will have a more accurate understanding of the impact on your service than before you made the change.

Assumptions

The final responsibility of engineering in the capacity-planning process is to highlight the design assumptions and any risks that are being taken with the product. It is very difficult to ship a product. It is even more difficult to get it out the door if you don't have any fundamental assumptions about how it will be used.

Did you build in caching at any level? Does it assume that traffic will be distributed in a certain way? Does the system break down if your assumptions become incorrect? Can you survive with 90% of your traffic going to a single Web page? What if it is evenly distributed over your entire corpus?

Did you assume a readers-to-writers ratio for your system? What if it suddenly flips and all of your traffic surges towards the more expensive of the two?

Did you assume that load would be distributed evenly over the day? What if you get a traffic spike from a media event or successful advertising?

One great thing about these assumptions is that clearly stating answers to them provides a warning for your product managers. However, if we take actions to invalidate these assumptions, we first need to do the engineering to survive it or consciously accept the risk it entails.

Risks

Through this analysis, we have identified a couple areas of risk that should be mitigated or accepted as part of the planning process. Any risk that is mitigated by capacity planning is an explicit tradeoff against money (or alternate uses for the resources aka opportunity cost). So the goal should never be to eliminate or even reduce risk. The goal should be to drive the system to the appropriate level of risk for the lowest cost.

Each of the theoretical minimum blow-up factors makes for a good place to start itemizing the places where you are spending money to avoid risk. What does that second datacenter buy you? How about your disk configuration or even the vendor for your hardware or cloud service? What problem are you avoiding, and how much are you avoiding it by? These problems and costs will be extremely specific to the service and might involve reliability, durability, performance, or even developer velocity. Or they could easily include all of them.

The other area of risk mitigation is around future growth: the thing that we are traditionally trying to estimate in capacity planning. You should plan at least as far ahead as the order time for your resources. This could be five years if you erect your own buildings, or 10 minutes if you run a small service in a cloud and have good automation and a credit card on file. Within that horizon, you must understand the risk of spending too much money, or of running out of capacity to handle your growth. That tolerance should define how aggressively you provision your service for growth.

PLAN B

Where would you be without a solid backup plan? Living in fear of a “success disaster” and failing to get a good night’s sleep. Success disaster can occur when your product becomes so popular that the number of users who show up overwhelms your ability to actually provide the service they seek. It is great to be wanted, but this can squander your one opportunity to make a good impression.

As part of this process, always take time to understand what would happen when your demand for capacity exceeds what is available. What if the service fails and there is no way to save it? You might want to highlight that clearly in the process. On the other hand, if you can come up with some ideas for either graceful failure modes or improvements to efficiency and estimate the time required to implement them, then you can sleep more soundly. You just need to be able to build and deploy those solutions quickly enough to help. I’d suggest setting some alert thresholds to signal when you really should start to panic.

Product

The job of the product managers in the capacity-planning process is simple: Create a product that users love so much that it completely obliterates the planning and, after a brief period of panic, brings tears of joy to the finance team and the rest of the company. No pressure.

But that isn’t the part of the job we are focusing on right now. This is about communication of the practical costs and risks that the engineers either have built or plan to build into the product, the needs of the users, and making sure those are aligned.

Alignment with Engineering

PROVIDE INFORMATION TO ENGINEERING

Start with the information you can provide to engineering. The big items are growth estimates, user behavior changes, and real product requirements. Attempt to understand how users will use the service in the future, with as much lead time as your capacity planning requires and in the growth dimensions used by the engineering model. These estimates will form the basis for future growth if you want to predict anything more complicated than an extrapolation from the past. You can pull the numbers out of thin air or dive deeply into the metrics of similar products, or survey your customers—whatever will provide you with numbers you can confidently use to drive planning far enough into the future that it makes a difference.

Second, you should help identify changes in user behavior, particularly when they conflict with any assumptions that have been made in the product design. Did you plan on building a system for sharing photos publicly, but people are using it to back up their receipts and keeping everything private? If so, you probably want to rethink that caching strategy. These kinds of changes can be critical to the success of your product but also need to be accounted for in the planning.

Third, product requirements should come from someone representing the customer. Is availability critical? Two nines or five? How fast does the system need to be? And if your answer is “all the nines” and “instant,” then you probably need to rethink how you see the product. The goal should be to identify at least a minimum level of quality (i.e., the minimum level before it slows adoption), or better yet, a range of requirements that can be tied to how customers will feel about the product. For many products, it is possible to run experiments, making changes to the performance of the system, and observing the behavior of users in order to get firm numbers around what the real requirements are. For example, you can increase latency artificially or decrease it by moving the user to a less loaded copy of your infrastructure and measure differences in how they use the product.

An example would be latency requirements and establishing their impact on customers. We should understand how users perceive our service based on different levels of responsiveness. This might let us learn that anything faster than 250 ms is indistinguishable to the user and that anything slower than 750 ms conveys a sense of low product quality. This would let us target between 250 and 750 ms as an ideal range for our planning.

You can use this kind of information to drive engineering and finance decisions, potentially making your product much less expensive to operate or much easier to develop and deploy. The earlier you can create and refine these numbers and feed them into the design process, the more potential you have to build the product you need at the lowest cost.

RECEIVE INFORMATION FROM ENGINEERING

The engineering assumptions about the product should provide extremely valuable information about how it is being designed and deployed. In a negative sense, it should highlight the parts of the product that are either expensive or particularly risky if the consumer behavior changes dramatically. So it is important to keep these in mind either while marketing or while designing upcoming features because these are likely to increase the risk in the system.

On a more positive note, you may discover functionality that is particularly inexpensive or trivial to implement in the product, and this may help you come up with features for the future. Or you might gain a better understanding of how your competitors may have designed their infrastructure, letting you focus on features that will be difficult for them to quickly emulate.

Finance

“Finance” is shorthand for people responsible for keeping the business funded and growing. At the end of the day, this is everyone’s responsibility, but most companies have people that focus on these types of things to the exclusion of developing new products or talking with users. Most importantly, this is the role that **looks across the entire company** and not just your product.

Asking Hard Questions

So the defining characteristic of Finance is actually that of scope, and with that comes the ability to make tradeoffs across multiple products and across time.

Working through these questions in a small well-funded company with a single product is a difficult task. As the size and complexity of the products offered by the company increase, doing holistic capacity planning becomes increasingly difficult. The goal should be to gain the advantages of scale and risk pooling to offset this increased challenge.

TIMELINES

Start by filling out a small table (Table 1). We can assume that we have a good resource model in place and this has been done before, but invite the engineering and product people to help generate these numbers.

Sum up those dates, and if any part of this can only be done at specific times of the month, quarter, or year, add that in as well. Specifically, if your organization has budgets that are only flexible at quarterly or annual boundaries, it leads to a substantial increase in your lead time. This lead time is critical for anyone doing capacity-planning to be familiar with.

One of the most valuable things that a company can do to drive down capacity-planning risk and cost is to shorten that cycle. So over time, each step should be evaluated to see whether it provides value to offset the cost of the additional delay.

Lead Time	Topic	Description
	Generate planning numbers	Create the numbers that drive the process
	Estimate resources	Turn growth estimates into specific resource requests
	Request resources	Ask for budget and equipment
	Approve resources	Complete budget and ordering process
	Provision resources	Deliver and set up
	Ready to serve	Provide service to users

Table 1: Timeline for resource delivery

CONFIDENCE AND PRECISION

For each growth estimate that you receive from each product, you need to understand more than just the bottom-line number around the resources required. The first thing to do is ignore the precision. Precise numbers are easy to generate since they just require the multiplication of two or three made-up numbers and very little rounding, but they trick almost everyone into thinking that they are “better” than a person who just writes down a 10 and moves on with their life.

Dig deeper. Check out the confidence associated with both the capacity model as well as the growth predictions that went into turning that model into a future-growth forecast. Check out the sensitivity that those estimates have to their time horizon and how far into the future people are being asked to forecast. There is a very natural tendency to overstate the potential upside of a launch. The people involved are likely very excited about the changes, and that may make it difficult for them to remain objective. Challenge these assumptions and make sure they aren’t unnecessarily keeping you from spending resources seeking out other opportunities.

Finally, look across the products and see whether they appear to be correlated with each other in terms of growth and cost. You may be able to collapse the “upside” of several products together and plan on having only some of them succeed. This is a very specific way to trade increased risk for the organization against lowered cost. It assumes that the resources you are under-planning are fungible across products and that someone is in a position to resolve conflicts if your demand outstrips your resources because of this choice.

Alternatively, there may be two forms of synergy that make this particularly dangerous. The first is technical coupling, where the success of one product forces work on the other products, so they aren’t actually independent. The second is that if one product is successful and is able to pull along the other products

indirectly, then growth may be correlated, again increasing the risk of a bundled approach to planning. Brand recognition, news coverage, cross-promotion between products, or any other ideas you have can tie together the growth rates of various products. In cases where you desire more traffic, these are great problems to have, but you must consider the potential in your planning.

PORTFOLIO RISK MANAGEMENT

The risks of each product and their growth scenarios must be understood in the context of the larger portfolio of the company. These risks come in two general forms: the specific product risks and the organizational risks.

The specific product risks should largely be as explained by the engineers. What happens if growth exceeds the capacity that is planned for the service? Do we have legal liabilities or customer dissatisfaction that will be particularly harmful to the company? Options may be available to mitigate these risks if it is not possible to provide the required resources, but they should be fairly explicit.

The second class is more difficult. Identify the risks that cross product boundaries and that may be less obvious to the specific product teams. If you have a company with multiple products that have dependencies on each other, this is where you need to look for those and highlight them specifically. Make sure that failure (or success) of one product doesn't do anything surprising and harmful to other products. This is when help from leaders within engineering will be very helpful to identify linkages and make them explicit. It may provide a sort of transitive priority or mutual dependencies between different products that need to be evaluated.

SUPPLY CHAIN AND LOGISTICS

Very specific to the table at the beginning of this section, it is important to understand what can be done to drive down the time required to go from having the desire to fund a product to having the ability to make that product functional with the resources in hand. With a cloud-provisioning model and a small-scale relative to your cloud provider, this may be trivial. But as your resource requirements increase in size or complexity, this may be about shaving months or even years off the system.

The rule of thumb here should be that **if you know what you are going to do in the future, you probably shouldn't get hung up on the paperwork**. This is much more difficult than it sounds and will likely present a challenge for any company that tries to implement it, but the goal is simple: shrink the time horizon between taking estimates and providing capacity. If the resources used by different products are fungible, you can pool them and manage their provisioning much more quickly than their full lead time.

Alternatively, if you spend the money before the customer arrives and you have reasonable fungibility between resources, you may be able to greatly reduce the time from request to provisioning, by ordering the resources in advance and scheduling the capacity plans to arrive in time for provisioning rather than for ordering the resources. An example of this would be building out datacenter space based on past growth trends for the company, but not deciding which product you would fund until right before the machines landed.

Fungibility is obviously very helpful: letting tradeoffs be delayed until the last moment, having resources shifted as necessary between different products, or keeping pooled resources available to manage risks on short notice.

RELIABILITY AND OTHER METRICS

The final valuable questions center around the metrics that each product is attempting to achieve. Understanding what these mean and the choices made by engineering to achieve them, as well as the value provided to the customer, is critical to providing the "best" experience possible at the lowest cost.

Reliability is the easiest example, since it is fairly straightforward to buy reliability with increasingly large piles of money as you request more "nines" of availability. However, in most cases, it actually has diminishing returns to your users. Take the time to dive into this for the big products, find commonalities around how you are reaching your targets, and look for the things that cost the most. The cost could be in buying Tier-4 vs. Tier-1 datacenter space, fault-tolerant hardware, licensed software solutions, or through engineering and operational complexity that slows down your rate of development. Don't underestimate the cost or value of having reliability designed into your software stack and your operational practices. It may be a much more effective investment than hardware. Having common solutions across the company and regular investigations into each of these choices can provide opportunities to improve products and save money at the same time.

Prioritization

The most difficult task that will come up at the company level is that of prioritization. In most organizations, it will almost certainly be impossible to fund the capacity requested of each product at its most optimistic growth rate without any improvements in efficiency. And over any reasonable period of time, it probably isn't a wise investment either. On the other hand, in case of small companies or startups, the cost of resources is probably small relative to other expenses. As a result, the limiting factors around prioritization won't be around capacity planning but engineering time or management attention.

What is important is having a full understanding of the risks of underfunding each product relative to its actual growth. This

Capacity Planning

risk is likely an “opportunity cost” in many cases as well as some more “real” costs in the places where customers are negatively impacted by the underfunded products. Prioritization, then, is about making clear choices between products so that they can operate with certainty, and doing it in a clear and timely manner. Timeliness is particularly critical in systems that have long cycle times since time spent in analysis is actually costly in terms of the accuracy of estimates and planning that fed into the process.

Conclusion

More than just drawing graphs of how services will grow in the future, capacity planning should ideally be a process that pulls together different parts of the organization to determine how resources should be allocated to maximize their benefit to the company. Out of this will flow improvements in engineering, product, and process in a virtuous cycle.



Calling All ;login: Readers!

We're looking for:

- * Programmers * Testers
- * Researchers * Tech Writers
- * Anyone Who Wants to Get Involved

Find out more by:

-- Checking out our Web site:
<http://www.freebsd.org/projects/newbies.html>

-- Downloading the Software:
<http://www.freebsd.org/where.html>

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!



The FreeBSD Community is proudly supported by:

The
FreeBSD
FOUNDATION

Help Create the Future Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

`/var/log/manager`

Daily Perspectives for the Sysadmin and the Manager

ANDREW SEELY



Andy Seely is the chief engineer and division manager for an IT enterprise services contract, and an adjunct instructor in the Information and Technology Department at the University of Tampa. His wife Heather is his PXE Boot and his sons Marek and Ivo are always challenging his thin-provisioning strategy. andy@yankeetown.com

Consider a city zoo on a clear blue-skies summer day. Animals to tend and children's field trip groups to marshal. Line up to pet the goats, then stop for a juice break. Underneath the zoo, a subway switching station. Full commuter trains barreling past at 80 miles per hour. Red lights, green lights, timing trains and tracks and stops and people and go-go-go in the pitch-dark tunnels and klieg-bright platforms. You can walk up the steps from the train platform and be at the zoo.

Two worlds in the same place. There's a connection between them, but these two worlds have very different functions and operate with different senses of priority and purpose. They function in different ways, but in the end they are part of the same system, and both have to work correctly for that system to be healthy.

Consider the IT workplace. Instead of zoo and subway station, there are sysadmins and technical managers. I'll leave it to the reader to decide which is best mapped onto which, but the same idea applies: different purposes and vastly different senses of pressure, yet both have to function correctly for the overall system to perform.

It's a Normal Day for a Sysadmin

Sysadmins are different everywhere you go. And good sysadmins are different from everyone around them. But they have some things in common. A sysadmin might start the day early, might start late, might work all day and all night, or might work for 48 straight hours and then sleep until the pager rings. The work flow probably looks like this: You have a ticket queue in RT, you have a mailbox full of alerts, or you have a list of all the things you have to do to keep everything running. If something's down, you work on that first. If someone's complaining, you work on that first. If someone's complaining and telling you in exceptional detail how they know how to do your job better than you, you work on that last. If you've got a lot of things to work on and they're all about the same priority, you work on the things you enjoy most, the things you can get done fastest, or you just alphabetize and start with whatever begins with "A." If you're a truly elite sysadmin, you might alphabetize in Klingon and work from there.

You might find yourself spending a day writing a shell script for 12 hours, just for fun. I remember a particular day very well back in 1997 when I spent 12 hours on a Solaris 2.5.1 box writing arrays, queues, and other such functions in pure `/sbin/sh` Bourne script. I even figured out an effective approach to do job control for subprocesses before I discovered `/bin/jsh`. Yes, I had `csh`, `bash`, and `Perl` handy and knew how to use them. But I had a project to finish and a reputation to uphold, and if you can't figure out how to write a decent associative array in Bourne, then what kind of sysadmin are you? Along the way, I rebooted servers, cleared file systems, restarted print queues, reset passwords, and took care of all the annoying little jobs as if I was waving flies off my Jello. It was a pretty typical day: long hours, lots of work, but I was achieving things no one else on my team was even close to capable of doing.

It's a Normal Day for a Technical Manager

Managers are the same everywhere, and I'm no different. On any random day: A high-profile internal user wants to know why his BlackBerry email takes an extra minute to sync, and demands an answer now. The quote for the new monitoring system is 50% over budget, and a project manager needs guidance (and a vendor needs to get put on notice). No fewer than three employees have had grandparents die this week and need to take time off, oh, and I need to arrange to send flowers to funerals. Vice presidents are demanding a report on why customer surveys are down five points in the last month and want to know whether this represents the trend of a failing team. I can predict that in six months I'm going to get asked to make a particular miracle happen, so I ask the team to do some preparatory work so that it won't have to be done in crisis-mode. Then I struggle for five and a half months to get anyone to do anything without force and ultimately face an operational disaster that I knew was coming. I've got budget to pay for all of 15 minutes of training for every employee for the year, if spread out "fairly," and I'll get to hear how the company doesn't "take care of people" when I have to politely and regrettably deny training that isn't directly related to our core task.

And I've got this sysadmin kid in love with esoteric tools spending 12 hours on what I know is only a two-hour job, waving me off like I'm a fly on Jello when I'm talking to him because he's "in the zone" with his Bourne shell "magic" that no one else will understand when he's done with it. At the end of any given day, I might have long hours, lots of work, and feel like I did nothing of any substance at all.

Crisis Day for a Sysadmin

It's all-hands-on-deck. Major system failure. The best sysadmins know their systems to the core. They understand the applications, the protocols, the operating systems (especially the operating systems!), the networks, and the security controls and boundary protections. Understanding what's going on, knowing where to look for the hung process, the full log file, the protocol error, the expired certificate, the incompatible peripheral device, the wonky DNS server that's giving a slow answer, but only for 25% of queries: a true sysadmin is like a magician when there's a major problem.

I fondly remember the time, back in 2007 or so, when I had a serious problem with machine-to-machine communications between servers A and B. Nothing had changed in the environment, yet the systems weren't communicating and the service was failing in very odd ways. I traced it down to a corrupt drift file on server C. I don't even remember the details, other than the fact that the overall system turned out to be very

sensitive to timing, and my Nagios had alerted on what turned out to be a second-order effect of a weird and really subtle NTP problem on a secondary host. When it's a serious problem, everything else stops and a sysadmin is the only one who can save the day.

Crisis Day for a Technical Manager

Keep the VP informed every hour. Keep the customer-facing group updated every 30 minutes. Coordinate people being called in. Prevent people from duplicating effort. Wait, don't we have to pay some of these people time-and-a-half? Do I have approval for that? Take the phone call from finance about the lost revenue while the service is down. Our reportable metrics are going to suffer. Did we get an RT ticket in? The other guys on the team aren't answering their phones and pagers, and we need reinforcements. I don't mean to say I told you so, but didn't I ask for a status report on this subsystem six months ago? Have to start on the briefing I'll have to give tomorrow to the VP or higher, maybe a briefing I'll give while standing up in front of his desk.

Ask the sysadmin for an update, and get waved off because this is hard stuff, you know, and I need to quit bothering him so he can focus. He's pounding away in four xterms at once and seems like he's chasing down a rabbit hole on a server that's not even part of the outage, and he seems completely unable to tell me what he's thinking. I really don't think he's looking in the right place, but there's no one else on the team who can match his skills. So...step back and hope.

We're Working Together—Really, We Are

If you're a rock-star sysadmin with a technical manager asking you for a status update on the fix-action, take a moment to consider that that manager might have been a rock star in his own right and is now having to depend on you. If you're a manager just trying to survive the day and keep your sysadmins in line and out of trouble, just remember, you were once just like them.

Both sides of this story have a role to play and both bring value to the situation. The lesson is to understand the motivations and the perspectives and to value the good and help each other work through the difficult as a team. This is true in both directions, but it's easy to lose patience, and then respect, for each other in a tense situation. Remembering where you came from and providing the right mix of understanding and guidance to others is not easy, but that's what makes an organization work effectively. I'm the manager, and that's my job.

Special thanks to my good friend Hugh Brown at OpenDNS for his suggestions on this column.



Buy the Box Set!

Whether you had to miss a conference or just didn't make it to all of the sessions, here's your chance to watch (and re-watch) the videos from your favorite USENIX events. Purchase the "Box Set," a USB drive containing the high-resolution videos from the technical sessions. This is perfect for folks on the go or those without consistent Internet access.

Box Sets are available for:

- » LISA14: 27th Large Installation System Administration Conference
- » OSDI '14: 11th USENIX Symposium on Operating Systems Design and Implementation
- » TRIOS '14: 2014 Conference on Timely Results in Operating Systems
- » USENIX Security '14: 23rd USENIX Security Symposium
- » 3GSE '14: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education
- » FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet
- » HealthTech '14: 2014 USENIX Summit on Health Information Technologies
- » WOOT '14: 8th USENIX Workshop on Offensive Technologies
- » URES '14: 2014 USENIX Release Engineering Summit
- » USENIX ATC '14: 2014 USENIX Annual Technical Conference
- » UCMS '14: 2014 USENIX Configuration Management Summit
- » HotStorage '14: 6th USENIX Workshop on Hot Topics in Storage and File Systems
- » HotCloud '14: 6th USENIX Workshop on Hot Topics in Cloud Computing
- » NSDI '14: 11th USENIX Symposium on Networked Systems Design and Implementation
- » FAST '14: 12th USENIX Conference on File and Storage Technologies
- » LISA '13: 27th Large Installation System Administration Conference
- » USENIX Security '13: 22nd USENIX Security Symposium
- » HealthTech '13: 2013 USENIX Workshop on Health Information Technologies
- » WOOT '13: 7th USENIX Workshop on Offensive Technologies
- » UCMS '13: 2013 USENIX Configuration Management Summit
- » HotStorage '13: 5th USENIX Workshop on Hot Topics in Storage and File Systems
- » HotCloud '13: 5th USENIX Workshop on Hot Topics in Cloud Computing
- » WiAC '13: 2013 USENIX Women in Advanced Computing Summit
- » NSDI '13: 10th USENIX Symposium on Networked Systems Design and Implementation
- » FAST '13: 11th USENIX Conference on File and Storage Technologies
- » LISA '12: 26th Large Installation System Administration Conference

Learn more at: www.usenix.org/boxsets

Practical Perl Tools

Give it a REST

DAVID N. BLANK-EDELMAN



David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 29+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

Believe it or not, there's a good reason that this column returns to the subject of Web API programming on a fairly regular basis. As time goes on, much of the work of your average, ordinary, run-of-the-mill sysadmin/devops/SRE person involves interacting/integrating with and incorporating services other people have built as part of the infrastructure we run. I think it is safe to say that a goodly number of these interactions take place or will take place via a REST-based API. Given this, I thought it might be a good idea to take a quick look at some of the current Perl modules that can make this process easier.

Nice Thesis, Pal

Before we get into the actual Perl code, it is probably a good idea to take a brief moment to discuss what REST is. People used to argue about what is and what isn't REST, but I haven't heard those arguments in years. (I suspect enough people abused the term over the years that those who cared just threw up their hands.) Back in 2012 I wrote a column that included an intro description about REST; let me quote from myself now:

REST stands for "Representational State Transfer." The Wikipedia article on REST at http://en.wikipedia.org/wiki/Representational_State_Transfer is decent (or was on the day I read it). Let me quote from it:

"REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource..."

Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: presented with a network of Web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

(That last quote comes from Roy Fielding's dissertation, which actually defined the REST architecture and changed Web services forever as a result.)

That's a pure description of REST. In practice, people tend to think about REST APIs as those that use a combination of HTTP operations (GET/PUT/DELETE) aimed at URLs that represent the objects in the service. Here's an overly simplified example just for demonstration purposes:

```
GET /api/shoes - return a list of available shoes
GET /api/shoes/shoeid - return more detailed info on that shoe
DELETE /api/shoes/shoeid - delete that kind of shoe
PUT /api/shoes/shoeid - update inventory of that kind of shoe
```

This just gives you a fleeting glance at the idea of using HTTP operations as the verb and intentionally constructed URLs as the direct objects for these operations. But to quote the Barbie doll, “APIs are hard, let’s go shopping!”

What Do We Need

As a good segue to the modules that are out there, let’s chat about what sort of things we might like in a module that will assist with REST programming. I’m focusing on REST clients in this column, but who knows, we might get crazy in a later column and talk about the server side of things as well.

The first thing we’ll clearly need is an easy way to construct HTTP requests. The messages being passed to and fro over these requests is likely to be in JSON format (the current *lingua franca* for this sort of thing) or XML format, so it would be great if that didn’t cause the module to break a sweat. Beyond this, it can be helpful to have the module understand the usual request-reply-request more workflow and perhaps add a little syntactic sugar to the process to make the code easier to read. Okay, let’s see what Perl can offer us.

HTTP Me

We are going to be looking at modules that do lots more hand-holding than this category, but I feel compelled to start with something a little lower level. Sometimes you will want to whip out a very small script that makes a few HTTP calls. The classic module for this sort of thing was LWP::Simple or LWP::UserAgent (as part of the libwww package). Recently I’ve found myself using two other modules instead.

The first is one of the `::Tiny` modules. You may recall from a previous column that I love that genre of modules. These are the relatively recent trend in Perl modules to produce a set of modules that are mean and lean and do one thing well with a minimum of code. The `::Tiny` module in play for this column is `HTTP::Tiny`. Here’s a small excerpt from the doc:

```
use HTTP::Tiny;

my $response = HTTP::Tiny->new->get('http://example.com/');
die "Failed!\n" unless $response->{success};

print "$response->{status} $response->{reason}\n";
...
print $response->{content} if length $response->{content};
```

As you can see, performing a GET operation with `HTTP::Tiny` is super easy (the same goes for a HEAD, DELETE, or POST) as is getting the results back. `HTTP::Tiny` will also handle SSL for you if the required external modules are also available. I’d also recommend you check out the small ecosystem of available modules that attempt to build on `HTTP::Tiny` (e.g., `HTTP::Tiny::UA`, `HTTP::Tiny::SPDY`, `HTTP::Retry`, and `HTTP::Tiny::Paranoid`).

Besides using `HTTP::Tiny`, I’ve also been enjoying using some of the fun stuff that comes with Mojolicious, the Web programming framework we’ve seen in past columns. For simple operations, it can look a lot like `LWP::UserAgent`:

```
use Mojo::UserAgent;

$ua = Mojo::UserAgent->new;
print $ua->get('www.google.com')->res->body
```

That’s not all that exciting. More exciting is when you combine this with some of the great Mojolicious DOM processing tools. Even more fun is when you use the “`ojo`” module to construct one-liners. (Quick explanatory aside: The module is called “`ojo`” because it gets used with the Perl runtime flag `-M` used to load a module from the command line. So that means you get to write `Mojo` on the command line.) Once again, let me borrow from the Mojolicious documentation to show you a couple of cool one-liners:

```
$ perl -Mojo -E 'say g("mojolicio.us")->dom->at("title")->text'
Mojolicious - Perl real-time web framework
```

This uses the `g()` alias to get the Web page at `http://mojolicio.us`, find the title element in the DOM, and print the text in that element (i.e., the title of the page).

```
$ perl -Mojo -E 'say r(g("google.com")->headers->to_hash)'
```

This code performs a GET of `google.com`, returns the headers it gets back as a hash, and then performs a data dump (`r()`) of them. The end result looks something like this:

```
{
  "Alternate-Protocol" => "80:quic,p=0.002",
  "Cache-Control" => "private, max-age=0",
  "Content-Length" => 19702,
  "Content-Type" => "text/html; charset=ISO-8859-1",
  "Date" => "Fri, 28 Nov 2014 03:56:53 GMT",
  "Expires" => -1,
  "P3P" => "CP=\\\"This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en
&answer=151657 for more info.\\\"",
  "Server" => "gws",
  "Set-Cookie" =>
  "PREF=ID=fdfedfe972efadfb:FF=0:TM=1417147013:LM=141714701
3:S=hy1EI4K1BJDEX3to; expires=Sun, 27-Nov-2016 03:56:53 GMT;
path=/; domain=.google.com, NID=67=kC0tIKopo0mStqWp3xSj7
nM0iQkt-Gol9D3Ena9y8Ecam95Z2Ki-c7-NGjWYG878nHQ6tVE-Y3
JkqAM68YR1B6IsGuDL2Cd4UCYI2N35VMM66RcywTTGo6hAH8_Al8Wq
; expires=Sat, 30-May-2015 03:56:53 GMT; path=/; domain
=.google.com; HttpOnly",
  "X-Frame-Options" => "SAMEORIGIN",
  "X-XSS-Protection" => "1; mode=block"
}
```

Practical Perl Tools: Give it a REST

I find the ease of working with the structure of the page (via the DOM or CSS selectors) to be particularly handy, but do check out the rest of the documentation for the many other neat tricks Mojolicious can perform.

Get On with the REST

So let's start looking at the sorts of modules that are trying to help us with our REST work. We'll take this in the order of least hand-holdy (is that a word?) to most hand-holdy. You'll find that the earlier modules look very much like the HTTP request modules we've already seen. For example, let's see some sample code that uses REST::Client. For almost all of the examples in this column, we are going to use the handy sample REST service the developer Thomas Bayer has been kind enough to provide (as a demo for his sqlREST package found at <http://sqlrest.sourceforge.net>).

```
use REST::Client;

my $rc = REST::Client->new(
    host => 'www.thomas-bayer.com',
    timeout => 10, );

$rc->GET('/sqlrest/CUSTOMER/');
print $rc->responseContent(),"\n---\n";
$rc->GET('/sqlrest/CUSTOMER/3');
print $rc->responseContent();
```

The result of running this code looks something like this:

```
<?xml version="1.0"?><CUSTOMERList xmlns:xlink=
"http://www.w3.org/1999/xlink">
<CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/0/">0</CUSTOMER>
<CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/1/">1</CUSTOMER>
<CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/2/">2</CUSTOMER>
<CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/3/">3</CUSTOMER>
...
---
<?xml version="1.0"?><CUSTOMER xmlns:xlink=
"http://www.w3.org/1999/xlink">
<ID>3</ID>
<FIRSTNAME>Michael</FIRSTNAME>
<LASTNAME>Clancy</LASTNAME>
<STREET>542 Upland Pl.</STREET>
<CITY>San Francisco</CITY>
```

We've performed a GET to receive a set of URLs in XML format that represent the available customer list and then performed a second GET to pull information for the customer with ID 3. This second step was all manually done (i.e., I picked #3 at random),

but you can easily imagine using something like XML::LibXML or XML::Simple to parse the initial list that was returned, and then use some complicated process to determine which customer ID (or all of them) for the second step.

So I bet you are wondering why REST::Client is any better than HTTP::Tiny in this case. It is only a hair more helpful. The helpful part comes largely in the new() call where we could set a default host (meaning we only have to put the path into the GET requests), a timeout, and settings for SSL/redirects (which we didn't use). Responses are a little easier to retrieve, but on the whole, nothing exciting.

A step up from this is something like WebService::CRUST. This module is a step further in the direction of "more hand-holdy" for a few reasons:

- ◆ It can take more "default" settings in the object constructor, so the actual query lines only have to contain the parameters explicit to the query.
- ◆ It knows how to hand the results back to a parser of some sort (i.e., to decode the XML or the JSON we get back).
- ◆ It adds some syntactic sugar, which makes the actual queries look more intuitive.

Let's do a quick rewrite of the REST::Client code to use WebService::Crust instead:

```
use WebService::Crust;
use Data::Dumper;

my $wc = new WebService::CRUST(
    base      => 'http://www.thomas-bayer.com/sqlrest/',
    timeout   => 10,
    # params => { appid => 'SomeID' },
);

# same as $wc->get('CUSTOMER')
# same as $wc->get_CUSTOMER();
my $reply = $wc->CUSTOMER;

print Dumper $reply->result;
print "\n---\n";
my $reply = $wc->get('CUSTOMER/3');
print Dumper $reply->result;
```

The first thing to note in this code is the constructor takes a base for the API so we never have to repeat the URL in our code. It also can take a params hash that will be used to add parameters to every call. For example, if your API required you to send along some API-specific key in each call, you could easily do it here. Our test service doesn't call for this, so I placed a commented-out version there instead. And, in the spirit of making the actual API calls easier, you can see that WebService::CRUST lets us write things as a method call (->CUSTOMER or get_CUSTOMER), which makes the code even more readable.

Now on to the more interesting part. If we were to actually dump out the contents of `$reply` at this point, we'd find it contained not the XML that the call returned, but a Perl data structure that represented the parsed version of that XML, hence the use of `Data::Dumper` to display it. The actual data structure looks like this:

```
DB<1> x $reply->result
0 HASH(0x7fb96cbdbe0)
  'CUSTOMER' => ARRAY(0x7fa1abd96e40)
    0 HASH(0x7fa1abc674b8)
      'content' => 0
      'xlink:href' => 'http://www.thomas-bayer.com/sqlrest/CUSTOMER/0/'
    1 HASH(0x7fa1abd82cc0)
      'content' => 1
      'xlink:href' => 'http://www.thomas-bayer.com/sqlrest/CUSTOMER/1/'
    2 HASH(0x7fa1abb29570)
      'content' => 2
      'xlink:href' => 'http://www.thomas-bayer.com/sqlrest/CUSTOMER/2/'
    3 HASH(0x7fa1a9ee7848)
      'content' => 3
      'xlink:href' => 'http://www.thomas-bayer.com/sqlrest/CUSTOMER/3/'
  ...
```

There's a hash with a single key called "result". The value of this key is a reference to a hash that contains a single key (CUSTOMER). That key has a reference to an array of hashes, each containing the XML elements we'll need to access. If you think this particular data structure is kind of icky, I'm in your corner. What you are seeing here is the default parse rules from `XML::Simple` at work. Sometimes they work well given a hunk of XML, sometimes not as well. They work better for the second query (the one where we request the info for customer #3):

```
{
  'LASTNAME' => 'Clancy',
  'FIRSTNAME' => 'Michael',
  'ID' => '3',
  'xmlns:xlink' => 'http://www.w3.org/1999/xlink',
  'CITY' => 'San Francisco',
  'STREET' => '542 Upland Pl.'
};
```

The two ways to deal with the icky data structure takes us too far afield to look at in depth, but just to give you a head start on the problem, you could either:

- ◆ write a subroutine that takes in the unpleasant data structure and returns one that is easier to use, or

- ◆ use the "opts" constructor option in the `new()` call to pass along options to `XML::Simple`. `XML::Simple` is quite willing to do your bidding, you'll just have to tell it exactly what you need.

Now, just so we don't lose track of one of the desired qualities of REST modules we mentioned earlier, I want to make sure JSON gets at least a brief mention. So far our test code has talked to APIs that return XML; what would we do if we had to talk to something that spoke only JSON? A couple of possibilities leap right to mind. First, we could switch modules. There are modules like `REST::Consumer` that behave similarly to `WebService::CRUST`. `REST::Consumer` offers a little less syntactic sugar than `WebService::CRUST`, but it does expect to receive (and send) JSON data as a default. Since I have a sweet tooth sometimes that craves the sugar (for readability purposes, I assure you), a second possibility is to continue using `WebService::CRUST`. It allows you to write:

```
my $wc = new WebService::CRUST(
    format => [ 'JSON::XS', 'decode', 'encode', 'decode' ]);
```

and from now on `WebService::CRUST` will speak JSON by using the `JSON::XS` (the faster JSON module) to decode and encode messages for you.

By the Way

As a way of winding down this column, I want to point out two other REST-related module types that may be interesting to you. The first takes the sugar part of the last section a wee bit further. There are modules like `Rest::Client::Builder` that let you inherit from them the capability to build OOP modules. In your module you spend a little time mapping out the API in your code and in return you get to write code that uses the API operations as if they were native calls. This is like the `->CUSTOMER` stuff from above only a little cleaner because you've been explicit up front.

The last module I want to show you is some combination of fun and debugging (or maybe debugging fun). The module `App::Presto` installs a command line tool called "presto" that provides an interactive shell for working with REST services. If you are used to debugging them using `CURL`, you may find that presto will make your life a little easier. Let's see a couple of quick sessions using it:

```
$ presto http://www.thomas-bayer.com/sqlrest/CUSTOMER/
http://www.thomas-bayer.com/sqlrest/CUSTOMER/> GET 3
{
  "STREET" : "542 Upland Pl.",
  "ID" : "3",
  "FIRSTNAME" : "Michael",
  "CITY" : "San Francisco",
  "xmlns:xlink" : "http://www.w3.org/1999/xlink",
  "LASTNAME" : "Clancy"
}
http://www.thomas-bayer.com/sqlrest/CUSTOMER/> quit
```

Practical Perl Tools: Give it a REST

Here's a JSON-based API session:

```
$ presto http://date.jsontest.com
http://date.jsontest.com> GET /
{
  "time" : "04:02:11 AM",
  "milliseconds_since_epoch" : 1417320131828,
  "date" : "11-30-2014"
}
```

Having a tool that lets you walk around an API like this can be mighty handy at times. And with that, let's bring this column to a close. Take care and I'll see you next time.

nsdi '15

12th USENIX Symposium on Networked Systems Design and Implementation

May 4–6, 2015 • Oakland, CA

NSDI '15 will focus on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

The program at this year's Symposium includes 42 refereed paper presentations on data centers, software-defined networking, wireless, data analytics, protocol design and implementation, virtualization, and much more.

The Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2015), taking place May 7–8, will be co-located with NSDI '15.

Register by April 13 and save!

www.usenix.org/nsdi15



Thinking about Type Checking

DAVID BEAZLEY



David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (<http://www.swig.org>) and Python Lex-Yacc (<http://www.dabeaz.com/ply.html>). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

A common complaint levied against Python (and other similar languages) is the dynamic nature of its type handling. Dynamic typing makes it difficult to optimize performance because code can't be compiled in the same way that it is in languages like C or Java. The lack of explicitly stated types can also make it difficult to figure out how the parts of a large application might fit together if you're simply looking at them in isolation. This difficulty also applies to tools that might analyze or try to check your program for correctness.

If you're using Python to write simple scripts, dynamic typing is not something you're likely to spend much time worrying about (if anything, not having to worry about types is a nice feature). However, if you're using Python to write a larger application, type-related issues might cause headaches. Sometimes programmers assume that these headaches are just part of using Python and that there isn't much that they can do about it. Not true. As an application developer, you actually have a variety of techniques that can be used to better control what's happening with types in a program. In this installment, we explore some of these techniques.

Dynamic Typing

To start, consider the following function:

```
def add(x, y):
    return x + y
```

In this function, there is nothing to indicate the expected types of the inputs. In fact, it will work with any inputs that happen to be compatible with the + operator used inside. This is dynamic typing in action. For example:

```
>>> add(2, 3)
5
>>> add('two', 'three')
'twothree'
>>> add([1,2], [3,4,5])
[1, 2, 3, 4, 5]
>>>
```

This kind of flexibility is both a blessing and curse. On one hand, you have the power to write very general-purpose code that works with almost anything. On the other hand, flexibility can introduce all sorts of strange bugs and usability problems. For instance, a function might accidentally “work” in situations where it might have been better to raise an error. Suppose, for example, you were expecting a mathematical operation, but strings got passed in by accident:

```
>>> add('2', '3')
'23'
>>>
```

Thinking about Type Checking

You might look at something like that and say “but I would never do that!” Perhaps, but if you’re working with a bunch of Web coders, you might never know what they’re going to pass into your program. Frankly, it could probably be just about anything, so it’s probably best to plan for the worst. I digress.

The lack of types in the source may make it difficult for someone else to understand code—especially as it grows in size and you start to think about the interconnections between components. As such, much of the burden is placed on writing good documentation strings—at least you can describe your intent to someone reading the source and hope for the best:

```
def add(x, y):
    """
    Adds the numbers x and y
    """
    return x + y
```

You might be inclined to explicitly enforce or check types using `isinstance()`. For example:

```
def add(x, y):
    """
    Adds the numbers x and y
    """
    assert isinstance(x, (int, float)), 'expected number'
    assert isinstance(y, (int, float)), 'expected number'
    return x + y
```

However, doing so typically leads to ugly non-idiomatic code and may make the code unnecessarily inflexible. For example, what if someone wants to use the above function with `Decimal` objects? Is that allowed?

```
>>> from decimal import Decimal
>>> x = Decimal('2')
>>> y = Decimal('3')
>>> add(x, y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in add
AssertionError: expected number
>>>
```

Alternatively, you might see a function written like this:

```
def add(x, y):
    """
    Adds the integers x and y
    """
    return int(x) + int(y)
```

This function will attempt to coerce whatever you give it into a specific type. For example:

```
>>> add(2, 3)
5
>>> add('2', '3')
5
>>> add('two', 'three')
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'two'
>>>
```

This too might have bizarre problems. For example, what if floats are given?

```
>>> add(2.5, 3.2)
5
>>>
```

Alas, the function runs but silently throws away the fractional part of the inputs. If that’s what you expected, great, but if not, then you have a whole new set of problems to worry about. Needless to say, it can get complicated.

Do type-related issues really matter in real applications? Based on my own experience, I’d answer yes. As a developer, you often try to do your best in writing accurate code and in writing tests. However, if you’re working on a team, you might not know every possible way that someone will interact with your program. As such, it can often pay to take a defensive posture in order to identify problems earlier rather than later. Frankly, I often think about such matters solely as a way to prevent myself from creating bugs.

Having better control over type handling in Python is mostly solved through techniques that add layers to objects and functions. For example, using properties to wrap instance attributes or using a decorator to wrap functions [4]. The next few sections have a few examples.

Managing Attribute Types on Instances

Suppose you have a class definition like this:

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

By default, the attributes of `Stock` can be anything. For example:

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.shares = 75
>>> s.shares = '75'
>>> s.shares = 'seventyfive'
>>>
```

However, suppose you wanted to enforce some controls on the shares attribute. One approach is to define shares as a property. For example:

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        'Getter function. Return the shares attribute'
        return self.__dict__['shares']

    @shares.setter
    def shares(self, value):
        'Setter function. Set the shares attribute'
        assert isinstance(value, int), 'Expected int'
        self.__dict__['shares'] = value
```

A property is a pair of get/set functions that captures the dot (.) operator for a specific attribute. In this case, all access to the shares attribute routes through the two functions provided. These two functions merely access the underlying instance dictionary, but the setter has been programmed to make sure the value is a proper integer. The resulting class works in exactly the same way as it did before except that there is now type checking on shares:

```
>>> s = Stock('IBM', 50, 91.1)
>>> s.shares = 75
>>> s.shares = '75'
Traceback (most recent call last):
...
AssertionError: Expected int
>>>
```

The verbose nature of writing out code for a property is a bit annoying if you have to do it a lot. Thus, if type checking is something you might reuse in different contexts, you can actually make a utility function to generate the property code for you. For example:

```
def Integer(name):
    @property
    def intvalue(self):
        return self.__dict__[name]

    @intvalue.setter
    def intvalue(self, value):
        assert isinstance(value, int), 'Expected int'
        self.__dict__[name] = value
    return intvalue

# Example
class Point(object):
    x = Integer('x')
```

```
y = Integer('y')
def __init__(self, x, y):
    self.x = x
    self.y = y
```

Here is an example of using the type-checked attribute:

```
>>> p = Point(2,3)
>>> p.x = 4
>>> p.x = '4'
Traceback (most recent call last):
...
AssertionError: Expected int
>>>
```

Alternatively, you can implement special type-checked attributes directly using a “descriptor” like this:

```
class Integer(object):
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        assert isinstance(value, int), 'Expected int'
        instance.__dict__[self.name] = value
```

A descriptor is similar to a property in that it captures the dot (.) operation on selected attributes. Basically, if you add an instance of a descriptor to a class, access to the attribute will route through the `__get__()` and `__set__()` methods. You would use the descriptor in exactly the same way the `Integer()` function was used in the above example.

Managing Types in Function Arguments

You can manage the types passed to a function, but doing so usually involves putting a wrapper around it using a decorator. Here is an example that forces all of the arguments to integers:

```
from functools import wraps

def intargs(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        iargs = [int(arg) for arg in args]
        ikwargs = { name: int(val) for name, val in kwargs.items() }
        return func(*iargs, **ikwargs)
    return wrapper

# Example use
@intargs
def add(x, y):
    return x + y
```

Thinking about Type Checking

If you try the resulting decorator, you'll get this behavior:

```
>>> add(2,3)
5
>>> add('2', '3')
5
>>> add('two', 'three')
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'two'
>>>
```

In practice, you might want to define a decorator that is a bit more selective in its type checking. Here is an example of applying type checks selectively to only some of the arguments. Note: This example relies on the use of the `inspect.signature()`, which was only introduced in Python 3.3 [1]. It will probably require a bit of careful study.

```
from functools import wraps
from inspect import signature

def enforce(**types):
    def decorate(func):
        sig = signature(func)
        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            for name, value in bound_values.arguments.items():
                if name in types:
                    expected_type = types[name]
                    assert isinstance(bound_values.arguments[name], \
                                    expected_type), '%s expected %s' \
                                    % (name, expected_type.__name__)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@enforce(x=int, z=str)
def spam(x, y, z):
    pass
```

In this example, the decorator works by obtaining the function's calling signature. In the wrapper, the `sig.bind()` operation binds the supplied arguments to argument names in the signature. The code that follows then iterates over the supplied arguments, looks up their expected type (if any), and asserts that it is correct. Here is an example of how the function would work:

```
>>> spam(1, 2, 'hello')
>>> spam(1, 'hello', 'world')
>>> spam('1', 'hello', 'world')
Traceback (most recent call last):
...
AssertionError: x expected int
>>> spam(1, 'hello', 3)
```

```
Traceback (most recent call last):
...
AssertionError: z expected str
>>>
```

A Word on Assertions

In these examples, the `assert` statement has been used to enforce type checks. One special feature of `assert` is that it can be easily disabled if you run Python with the `-O` option. For example:

```
bash % python -O someprogram.py
```

When you do this, all of the asserts simply get stripped from the program—resulting in faster performance because all of the extra checking will be gone. This actually opens up an interesting spin on the type-checking problem. If you have an application that executes in both a staging and production environment, you can do things like enable type checks in staging (where you hope all of the code is properly tested and errors would be caught), but turn them off in production.

There is also a global `__debug__` variable that is normally set to `True`, but it changes to `False` when `-O` is given. You might use this to selectively disable properties. For example:

```
class Point(object):
    if __debug__:
        x = Integer('x')
        y = Integer('y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

The Future: Function Annotations?

The future of type checking may lie in the use of function annotations. First introduced in Python 3, functions can be annotated with additional metadata. For example:

```
def add(x:int, y:int) -> int:
    return x + y
```

These annotations are merely stored as additional information. For example:

```
>>> add.__annotations__
{'return': <class 'int'>, 'x': <class 'int'>, 'y': <class 'int'>}
```

To date, the use of function annotations in practice has been somewhat scanty. However, projects such as `mypy` [2] have renewed interest in their possible use for type checking. For example, here is a sample function annotated in the style of `mypy`:

```
def average(values: List[float]) -> float:
    total = sum(values)
    return total / len(values)
```

A recent email posting from Guido van Rossum indicated a renewed interest in using annotations for type checking and in adopting the mypy annotation style in particular [3]. Standardizing the use of annotations for types would be an interesting development. It's definitely something worth watching in the future.

References

[1] <https://www.python.org/dev/peps/pep-0362> (Function Signature Object).

[2] <http://mypy-lang.org>.

[3] <https://mail.python.org/pipermail/python-ideas/2014-August/028618.html>.

[4] "Python 3: The Good, the Bad, and the Ugly" explains decorators and function wrappers: <https://www.usenix.org/publications/login/april-2009-volume-34-number-2/python-3-good-bad-and-ugly>.



Do you know about the USENIX Open Access Policy?

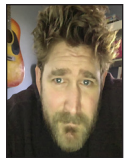
USENIX is the first computing association to offer free and open access to all of our conferences proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your financial support plays a major role in making this endeavor successful.

Please help to us to sustain and grow our open access program. Donate to the USENIX Annual Fund, renew your membership, and ask your colleagues to join or renew today.

www.usenix.org/annual-fund

iVoyeur Spreading

BY DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato .com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

In engineering, we are told to avoid repeating ourselves [1], but as a blogo-vangelizer (or whatever it is I'm doing now), I find it an increasingly burdensome and self-defeating mantra. It'd be great if I could give one talk and consider the subject of that talk closed. However, over the course of my first year as a developer evangelist, wherein I've delivered 12 conference talks, I've slowly begun to realize two very interesting facts.

First, most of the people who came to the conference don't see my talk. Even if the conference is a single-track, many attendees are consumed by a fire at work or by some really interesting "problem solving" (read: cat gifs), or they're in the hallway talking to the speaker from the last session. Whatever the reason, only a fraction of the attendees actually attempt to parse my one-two punch of words and slides.

Second, I very often fail to convey what I intend to the fraction of attendees who actually listen to me. I know this because when I talk to people who attend my talks, our conversations often go something like this:

Attendee: "Hey, I really enjoyed your talk."

Me: "Awesome, thanks! I hope it helped."

Attendee: "It did! I'm going straight home to <do horrifyingly wrong thing>."

Me: "Good god, why?!"

Attendee: "Well, silly, because you said <understandable but horrifyingly wrong interpretation of thing I said that would take me days to unravel and correct>."

Me: "Yeah, I can't take the credit for that. I actually copied it from <person who works at Microsoft>."

My point is, repeating yourself in an education context is not a bad thing (especially if you can't seem to get it right the first time). Many tech speakers riff on variations of the same talk over and over again for years. I used to suspect this was laziness, or that they'd gotten trapped by their own cult-of-personality, but now I'm realizing that you have to repeat yourself a lot to actually reach a critical-mass of mind-share in this medium. This is good news for me, because it's pretty often the case when I find myself belaboring a point—writing and talking a lot about the same subject—that it's because I'm trying to share something I wish I would have understood years ago.

Lately, I've been writing a lot about fat data points, which is the data storage format employed by Librato in our metrics product, and it's certainly the case that I wish I'd have understood them years ago. At Librato, a common use case for us is that of service-side aggregation. This is the practice of customers emitting measurements to us directly from inside worker threads running across lots and lots of geographically dispersed computers.

If a customer spawns ten thousand worker threads, and each of them emits a few measurements, we can easily wind up with upwards of fifty thousand in-bound data points in the

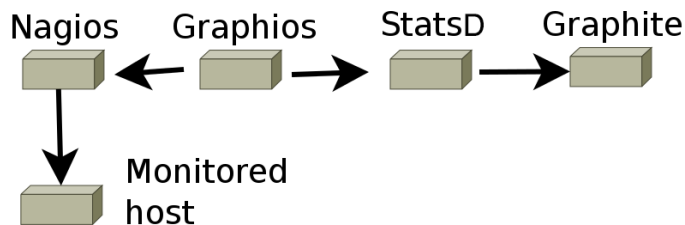


Figure 1: My tool-chain for the purposes of this article

space of a second, which we then need to aggregate in a statistically significant way. Taking the average of a set this size almost certainly destroys the truth hidden within the data, so for this, and many other reasons [2], we use fat data points to preserve the truth.

When writing about fat data points became talking about them at LISA14 [3], I got a pretty awesome question from Doug Hughes. It was simple, direct, and conveyed a deeply satisfying sense that I'd managed to successfully communicate the concept. Specifically, Doug's question was: "Okay, but how can WE use this?"

Avoiding the obvious (and correct) answer, that you should replace whatever you're currently using with Librato as soon as possible, it's actually possible to preserve spread data today with systems like RRDtool and Graphite. So in the interest of giving a meaningful answer to Doug's question, I'd like to show you how you'd configure Graphite to preserve spread data—the sum, count, min, max, average, and etc.

For the purposes of this how-to, I'm using a Nagios system that's emitting metrics to Graphite by way of StatsD. The metrics-extraction from Nagios is being performed by Graphios [4]. I'm going to use the one-minute CPU load metric as my example since I'm lazy and unimaginative. Figure 1 is a quick-and-dirty sketch of my setup.

Graphite controls rollups with the `storage-aggregations.conf` file. When a new metric is discovered for which there is no existing Whisper database, Carbon attempts to match the metric name against the rules in `storage-aggregations.conf`, beginning at the top and continuing to the bottom. The first line that matches the metric name wins, and no further lines are parsed once a match is found. If you're really paying attention, then you've probably realized that these rules make it impossible to assign different consolidation functions to different archives inside a Whisper file.

In order to maintain, for example, both the min and max values for a series in Graphite, therefore, we need to feed Graphite the same metric with two different names. That way we can match each variation of the metric name to a different rule in `storage-aggregations.conf`.

One simple way to do this is via StatsD's `*timer*` data type [5]. StatsD timers are intended to time things like function calls, to see how long they take to execute, but in practice you can use a timer to measure anything you might otherwise use a `*gauge*` to measure. The primary difference is that where passing a gauge into StatsD will merely result in a single value, a timer will cause StatsD to compute and emit a whole slew of interesting summary metrics, including the min, max, sum, count, and even percentiles for the StatsD flush interval.

So my strategy here is to emit the one-minute CPU load as measured by Nagios into StatsD as a timer. Then I'll configure storage-aggregation rules in Graphite to match the min, max, sum, and count for the summary statistics emitted by StatsD. When I'm done, I'll have different Whisper databases for this metric for each of the summary types I want.

Beginning in the Nagios configs, I'll configure a custom object variable called `*metrictype*` in the service definition of the metric I want to preserve spread data for:

```

define service{
    use                generic-service
    host_name          awacs
    service_description LOAD
    check_command      check_load!50,60,70!80,90,100
    _graphiteprefix    Piegan-Nagios
    _metrictype        timer
}
  
```

Graphios will parse out the `_graphiteprefix` and `_metrictype` custom variables, appending my prefix to the metric name, and translating the "timer" keyword into the associated StatsD wire-protocol [6]. On my system (hostname: `awacs`), this is what Graphios puts on the wire for StatsD:

```
Piegan-Nagios.awacs.load1:0.080!ms
```

No special configuration is required for StatsD. By default, StatsD will prepend two additional prefixes to my metric name: `stats` and `timers`. Here's what StatsD puts on the wire for Carbon:

```

stats.timers.Piegan-Nagios.awacs.load1.sum 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.sum_90 0.080
1416803719
stats.timers.Piegan-Nagios.awacs.load1.lower 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.upper 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.upper_90 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.sum 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.sum_90 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.count 1 1416803719
stats.timers.Piegan-Nagios.awacs.load1.count_ps 1 1416803719
stats.timers.Piegan-Nagios.awacs.load1.mean 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.median 0.080 1416803719
  
```

iVoyeur: Spreading

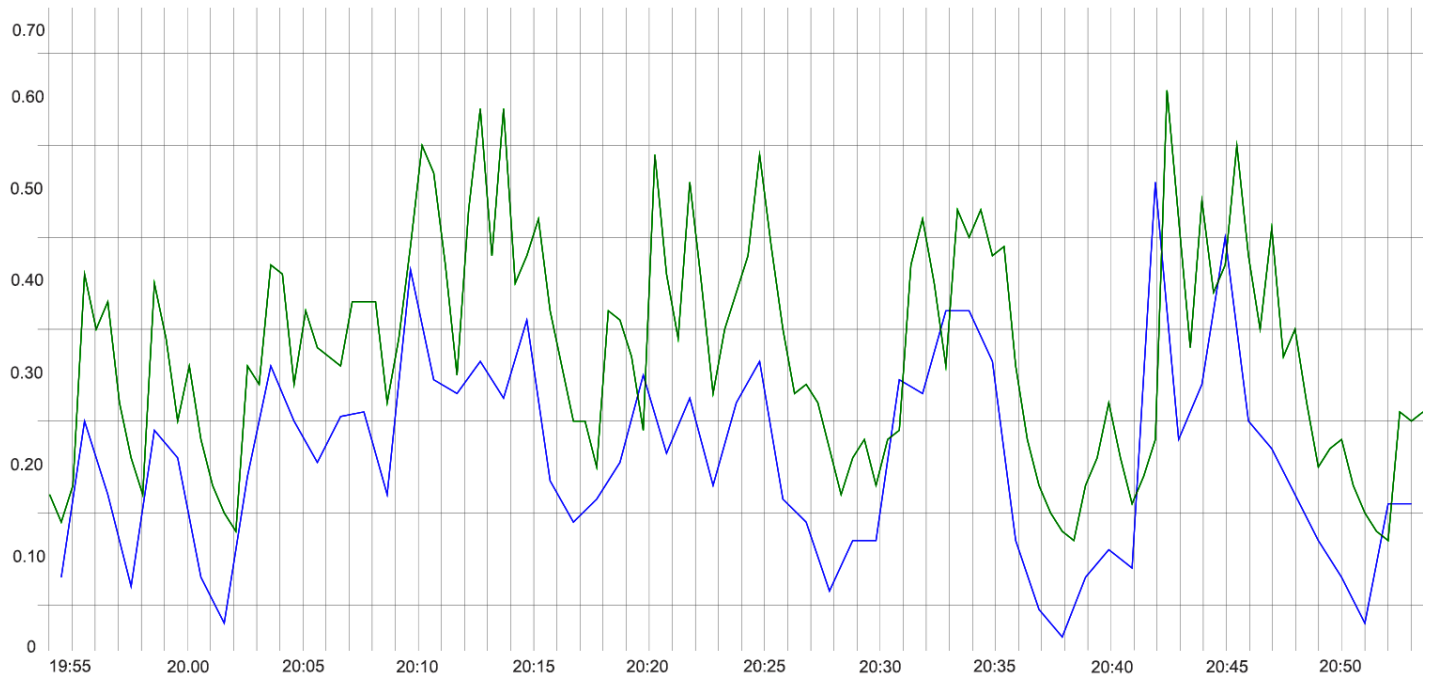


Figure 2: Plotting average vs. max for the same metric

To be clear, what’s happening here is StatsD is accepting the load1 metric, and, because we’ve specified that it is a timer (the “|ms” suffix emitted by Graphios), StatsD automatically computes all of these summarization metrics across its flush interval. Most of these are self-explanatory; the metrics that look like thing_90 are the 90th percentile for thing (i.e., it is a number that 90 percent of the measurements in the flush interval are less than). Count_ps is the count divided by the number of seconds in StatsD’s flush interval (literally, ps here stands for per second).

Moving to the Graphite side, I’ve added rules to match each of these StatsD summary metrics to my /opt/graphite/conf/storage-aggregations.conf file:

```
[min]
pattern = stats.timers.*lower$
xFilesFactor = 0.9
aggregationMethod = min

[max]
pattern = stats.timers.*(upper|upper_90)$
xFilesFactor = 0.9
aggregationMethod = max

[sum]
pattern = stats.timers.*sum$
xFilesFactor = 0.9
aggregationMethod = sum

<snip>
```

Carbon will use this file to properly create the Whisper databases for these metrics in a way that properly aggregates the data over time, preserving what’s important to us. I can verify it’s working by checking the creation log:

```
23/11/2014 04:43:26 :: new metric
stats.timers.Piegan-Nagios.awacs.
load15.upper_90 matched aggregation schema max
```

Or by running whisper_info directly against the DBs:

```
root@precise64# for i in *; do
> echo ${i}: $(whisper-info ${i} | grep aggre) ; done
count_ps.wsp: aggregationMethod: count
count.wsp: aggregationMethod: count
lower.wsp: aggregationMethod: min
mean_90.wsp: aggregationMethod: average
mean.wsp: aggregationMethod: average
median.wsp: aggregationMethod: average
std.wsp: aggregationMethod: max
sum_90.wsp: aggregationMethod: sum
sum.wsp: aggregationMethod: sum
upper_90.wsp: aggregationMethod: max
upper.wsp: aggregationMethod: max
```

At this point, perhaps obviously, I can craft a graph depicting the difference between the average and max rollups (Figure 2).

An interesting side effect of using StatsD timers this way is that you can also set up custom storage schemas for different types of spread data. For example, you could keep 10-second resolution on the mean and median values for 24 hours, and toss them after that while preserving the count and sum metrics at 10-minute and one-hour resolutions for years (since those rollups are effectively lossless and enable you to accurately compute the average at display time using the `divide()` function).

With a little thought, you'll wind up with a metrics storage system that far more accurately reflects your data, while making very effective use of space on disk. As always, I hope you found this useful in your quantification endeavors, and I highly recommend the use of spread data to protect the long-term fidelity of your beloved measurements.

Take it easy.

References

- [1] "Don't Repeat Yourself": http://en.wikipedia.org/wiki/Don%27t_repeat_yourself.
- [2] "Sensical Summarization for Time-Series": <http://blog.librato.com/posts/time-series-data>.
- [3] LISA14: <https://www.usenix.org/conference/lisa14>.
- [4] Graphios: <https://github.com/shawn-sterling/graphios>.
- [5] StatsD Metric Types: https://github.com/etsy/statsd/blob/master/docs/metric_types.md.
- [6] StatsD Line Protocol: <https://github.com/etsy/statsd/>.



Become a USENIX Supporter and Reach Your Target Audience

The USENIX Association welcomes industrial sponsorship and offers custom packages to help you promote your organization, programs, and products to our membership and conference attendees.

Whether you are interested in sales, recruiting top talent, or branding to a highly targeted audience, we offer key outreach for our sponsors. To learn more about becoming a USENIX Supporter, as well as our multiple conference sponsorship packages, please contact sponsorship@usenix.org.

Your support of the USENIX Association furthers our goal of fostering technical excellence and innovation in neutral forums. Sponsorship of USENIX keeps our conferences affordable for all and supports scholarships for students, equal representation of women and minorities in the computing research community, and the development of open source technology.

Learn more at:
www.usenix.org/supporter

For Good Measure Cyberjobsecurity

DAN GEER



Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

When you see something that is technically sweet, you go ahead and do it and you argue about what to do about it only after you have had your technical success. That is the way it was with the atomic bomb.—*Julius Robert Oppenheimer*

Automation exists to remove costs. Some costs are measured in money lost. Some costs are measured in inaccurate results. Some costs are measured in risk taken on. For any cost, however, the first question to ask is, from whose vantage point is such and such a cost a cost? Is it the person shelling out money that could have been saved? Is it the person receiving inaccurate outputs that drove needlessly poor decisions? Is it the person trading short-term convenience for long-term risk? Or is it the counterparties to each of those persons?

The best, most careful observers are now singing the same chorus, that automation is moving beyond the routinizable to the non-routine by way of the tsunami of ever bigger data. As such, it is not the fraction of people who are unemployed that matters; it is the fraction of people who will soon be unemployable. Machines that are cheaper than you, that make fewer mistakes than you, that can accept any drudgery that risk avoidance imposes are coming.

What does that have to do with cybersecurity and its measurement? Cybersecurity is perhaps the most challenging intellectual profession on the planet both because of the rate of change and because your failure is the intentional work product of sentient opponents. Can automation help with that? Of course and it already is, as you well know regardless of your misgivings about whether anomaly detection will work in an ever more “personalized” Internet—one man’s personalization is another man’s targeting.

The U.S. Bureau of Labor Statistics reports [1] that the five occupations with the best outlook for new jobs over the next 10 years are personal care aides, registered nurses, retail salespersons, home health aides, and food preparers/servers, with an aggregate 10-year employment growth of 2,388,400 at \$30,048 average income. Accepting that it takes 125,000 new jobs/month to hold unemployment steady, those five occupations can cover 19 of the next 120 months. On the world scale, those are good jobs—\$30,048 and you’re in the world’s top 6% [2].

High-paying jobs are precisely the ones that automation wants to take. Turning to BLS data for “information security analysts” [3], there are 75,000 of those with mean income of \$86,070 per year, putting ISAs in the top 0.5% on the world scale. The growth in that occupation for the coming decade is 37% (3.5% per year, the 16th best of all U.S. occupations), and of the 20 jobs with the fastest growth, only physicians’ assistants have a higher mean salary than ISAs. *Computerworld’s* survey [4] confirms the pinnacle status of information security practitioners, putting a CSO in the world top 0.2%.

So is automation gunning for the ISA role? If not, is it because ISAs are too few to bother with or is it that the job is too hard to automate (yet)? Shosana Zuboff’s [5] three laws bear repeating:

- ◆ Everything that can be automated will be automated.
- ◆ Everything that can be informed will be informed.
- ◆ Every digital application that can be used for surveillance and control will be used for surveillance and control.

Universities and the White House argue that as machines take over existing jobs, new opportunities are created for those who “invest in themselves.” As Federico Pistono argues [6] with clear numbers, that is not true. Ranking U.S. jobs by how many people hold them, computer software engineer is the only job created in the last 50 years with over a million job holders. It is #33 on the list; there are twice as many janitors. The most numerous job, delivery driver, is being automated out of existence as we speak. If cybersecurity jobs are safe from automation, should we be retraining all the truck drivers who are about to be unemployed as information security analysts? Are we lucky that our jobs come with sentient opponents? Are sentient opponents our job security—the source of both our pain and our power?

We cybersecurity folk are not the best paid. All but one of the 15 best paying jobs are in medicine (that one is CEO at #11), but as C.G.P. Grey [7] points out, once electronic health records really take hold, most of health care can be automated—at least the parts for diagnosis, prescribing, monitoring, timing, and keeping up with the literature.

But if it is true that all cybersecurity technology is dual use, then what about offense? Chris Inglis, recently retired NSA deputy director, remarked that if we were to score cyber the way we score soccer, the tally would be 462-456 twenty minutes into the game [8], i.e., all offense. I will take his remark as confirming at the highest level not only the dual use nature of cybersecurity but also confirming that offense is where the innovations that only Nation States can afford is going on. Put differently, is cybersecurity as a job moving away from defense toward offense insofar as the defense side is easier to automate? That won't show up in any statistics that you or I are likely to find; offense does not publish.

In sum, everything I see in the security literature and/or the blogosphere argues for automating cybersecurity. One must then ask if, in truth, our job description is to work ourselves out of a job. Or do we say that with a wink and a nod [9]?

References

- [1] Occupational Outlook Handbook: www.bls.gov/ooh/most-new-jobs.htm.
- [2] Wealth calculator (adjusted for purchasing parity) <http://www.worldwealthcalculator.org/results>.
- [3] Information Security Analysts: www.bls.gov/ooh/computer-and-information-technology/information-security-analysts.htm.
- [4] IT Salary Survey: www.computerworld.com/category/salariesurvey2014/.
- [5] “Skilled workers historically have been ambivalent toward automation, knowing that the bodies it would augment or replace were the occasion for both their pain and their power.” *In the Age of the Smart Machine (Basic Books, 1988)*, p. 56.
- [6] *Robots Will Steal Your Job, But That's OK* (CreateSpace, 2012), Chapter 9: www.robotswillstealyourjob.com/read/part1/ch9-unemployment-tomorrow.
- [7] “Humans Need Not Apply”: www.youtube.com/watch?v=7Pq-S557XQU.
- [8] Chris Inglis, confirmed by personal communication.
- [9] “Never write if you can speak; never speak if you can nod; never nod if you can wink.”—Martin Lomasney, Ward Boss, Boston.

/dev/random Smarter-Than-You Storage

ROBERT G. FERRELL



Robert G. Ferrell is a humorist, fantasy and science fiction novelist, and owner of the last two cats in the known universe who have never been featured on Youtube. rgferrell@gmail.com

Recently I had the honor of serving as a member of a teaching and volunteer examiner team for an amateur radio licensing class/exam. One of the lessons in the class went over exponent prefixes, from yocto- to yotta-. I mentioned to the students that giga-, tera-, peta-, and exa-, while virtually unknown to non-scientists two decades ago, had now entered into the general parlance as a result of developments in digital memory and storage technologies.

Heretical though it may sound, I have begun to question the requirement for these absurdly capacious storage devices. Do we really need enough storage, eventually, to house the complete thermodynamic history of every atom on Earth? The problem, as I see it, is that we've been seduced by the availability of cheap storage and so have lost any filters for what we deem worthy of retention. Why delete something when you can save it, just in case it might be useful to someone somewhere within the next 50 years? We have become, as a society, data hoarders.

You know what a hoarder is, right? That usually elderly lady or gentlemen who lives at the end of the block and who can't bear to throw anything away, with the result that their house is so completely filled with every conceivable item of useless junk that even first responders can't get in to rescue them when the need arises? I know about hoarders firsthand because there is a borderline example in my own family.

What makes hoarding a pathological condition is the complete and utter lack of discrimination. No filters whatever. I can see that same disease state germinating in the storage industry. Even Google Mail asks you, "Why delete anything?" Why, indeed. You certainly don't want to part with the 250 megabytes of ads that you receive annually for products that have no conceivable role in your life. And all those emails you got notifying you that people you've never even met in person have labeled you a moron in an online forum for taking a position on some current event topic that differs from their own? Keepers, for sure.

Now, if the only wildly extraneous crap being retained was by individuals with no sense of what actually matters in life that would be one thing, but I strongly suspect the affliction has overtaken corporations and governments, as well. If not, we wouldn't need exabytes of storage. *Exabytes*. Think about it. One with 18 zeroes. 1,000,000,000,000,000,000 bytes. The Earth itself contains roughly 9×10^{49} atoms. The way we're going, it probably won't be too long before we can store the spin state for every one of those fermions, bosons, and atomic nuclei (which of course would include the atoms of the storage system itself. Hello, recursion). But, why would we need to do that? "Just because we can" is a spurious, if not bordering on insane, rationale.

I wrote a fantasy novel in which mage-scientists had worked out a way to store a complete mental template for a human being in a crystal kept in a region of temporal stasis until needed. When you've grown tired of your body and its attendant aches and pains, or have

made some serious mistakes and want to take a Mulligan, you just grow an empty shell and dump that image into it. Presto! Back to being 25 again. Works well in a fantasy novel, not so much in the real world.

In that (varicose) vein, let's say for the sake of discussion that the human brain contains 10^{26} atoms. Most of that is hydrogen and oxygen, i.e., water, however, so we'll just stick with the 86 billion or so neurons themselves. There are 10 times as many glial cells, but we aren't entirely certain what role they play in cognition and memory, so we'll ignore them, too. All in all, there are at least 100 trillion (10^{14}) synapses present in a typical brain because each neuron can form thousands of links with other neurons. Every possible synapse, again simplifying for the sake of argument, can be either on or off, so we can represent that condition in a binary format. It would therefore require about 12 terabytes to store a snapshot of a human brain's wiring (presuming 8 bits to the byte). This process is complicated by the fact that human neural topology is plastic, making any attempt to capture it merely a discrete representation of a continuous process, but it's what we have to work with in this thought experiment (which has probably used up a few million synapses itself, if you're paying attention).

An exabyte, then, contains *ten thousand* times more information than a human brain could process at any given moment, even presuming that every single neuron could be devoted to the task, which is of course not a realistic proposition given that we need some not-insignificant number of CPU cycles for consciousness and sending each other lolcats. So, why do we experience this deep compulsion to have that much data thousands or millions of times over at our calloused fingertips? Is this a rhetorical question? What is the sound of one bit flopping?

"Big Data" is quickly evolving into "Incomprehensibly Huge Data." The day will come in the not-too-distant future when we will be completely removed from data processing, by necessity. Only computers will be able to access, munge (Hi, munge), and spit out this unimaginably huge pool of ones and zeroes. They won't even need us around to input anything with all of the SCADA and other automatic data-gathering mechanisms in place. I, Robot; you, extraneous.

My personal adaptation of artificial intelligence to this problem would be what I will dub "Smarter-than-you Storage." By this I mean storage devices that understand what actually matters and quietly discard everything else. You'll never know what data got tossed, of course, because the devices are programmed to do their thing without notifying anyone on the presumption that you, the human, are simply incapable of making those decisions rationally. I think we as a species have already demonstrated that. We archive everything, no matter how asinine or puerile: even data that an alien prosecuting attorney might well use as evidence of our non-sentience in some galactic competency hearing. The insane popularity enjoyed by videos of celebrities with no known talent wriggling their exposed posteriors leaps to mind. Most of the time what is trending on the Internet is cumulative idiocy.

Incidentally, if you take issue with my numbers or premise in the above diatribe, guess what? I got them from the Internet, our collective non-discriminatory storage farm. Thanks for bolstering my argument. I owe you one. Give me your email address and I'll send you a video of my cat chasing an invisible bug.*

*Offer not valid in Newtonian space.

Book Reviews

MARK LAMOURINE AND RIK FARROW

Becoming a Better Programmer

Pete Goodliffe

2014, O'Reilly Media Inc., 2014; 432 pages

ISBN 978-1-491-90553-1

Reviewed by Mark Lamourine

Peter Goodliffe subtitles his book "A handbook for people who care about code." It's obvious that he is passionate about writing with a clear and concise style. The book is a collection of tips and advice that I might give to new coders, and there's certainly nothing I would disagree with (although I looked for a section on suppressing the urge to cleverness and didn't find it).

He doesn't stop at code, and this is also good. He goes on to talk about the process and the personal side of a life of software development. The final section is more about interacting with the people who surround us than about any technical skills.

While I didn't find anything I disagree with, I also didn't find anything really actionable for me. This is a really good book to help someone who's just started coding get a perspective on the process as a whole. It's very easy for a junior person to focus on the language features and the rest of the tech. Goodliffe reminds them that there's more to the software development life than the editor and the compiler.

Ethernet: The Definitive Guide, 2nd ed.

Charles E. Spurgeon and Joann Zimmerman

Copyright 2014 O'Reilly Media, 2014; 484 pages

ISBN 978-1-449-36184-6

Reviewed by Mark Lamourine

I'm actually not sure to whom I'd offer this book; part history, part technical reference, part deployment guide, it's hard to categorize. I guess I'll have to talk through it and see.

Someone just beginning to work with Ethernet, either physically or by programming network protocol interactions, would be interested in the first section. Here, the authors tell the story of the initial development and then the evolution of Ethernet specifications and implementations. They spend a fair amount of time on CSMA/CD, which, while interesting, is unlikely to be found in production these days (I hope). More common will be the full-duplex twisted pair that follows, and the auto-negotiation protocols. The authors also cover the evolution of Layer 2 signaling, which introduces and makes up an Ethernet frame on the wire.

The second section is definitely for the reader who expects to handle Ethernet cabling and interface devices. In this section

Spurgeon and Zimmerman detail each of the existing Ethernet specs, although only the IEEE specs are still relevant. Again, the history is a useful base for understanding the current state. They also talk about the 10BASE40G and 100G specs that are not yet in production and which certainly will be limited to datacenters or to short-haul links between datacenters in an organization.

In the third section, they talk about the issues you'd face if you were building out an office or cube space as well as the structured cabling and termination within a datacenter.

The authors move on to a treatment of the networking hardware that binds the physical and logical networks together and then close with a section on network troubleshooting concepts and tips.

I enjoyed reading this book and would recommend it to anyone who expected to start supporting a datacenter network or a large desktop network space. Although I'd be hesitant to hire someone to do this work for me if I knew their only source of knowledge was a chapter from a book, this is certainly a good guide for someone learning under supervision. It also contains good information for someone trying to evaluate a set of proposals.

In the end, I'd say this is a good general reference. It has relevance for people working in or with Ethernet networks at any scale. I don't do that kind of work anymore, but I might still keep this book handy.

The Book of PF, 3rd ed.

Peter N. M. Hansteen

No Starch Press, 2014; 221 pages

ISBN 978-1-59327-589-1

Reviewed by Rik Farrow

PF is the packet filtering language used by OpenBSD, as well as FreeBSD, NetBSD, and Dragonfly BSD. I have the second edition of this book, but software and operating systems continue to evolve. There are now more shortcuts in the PF rules, including helpful ones, like including passing of packets in the same statement containing a redirect rule. ALTQ, the BSD traffic queueing system, has been replaced with a new traffic shaping system. Bridging has been added, allowing you to build firewalls with no IP addresses.

Hansteen has an easy-to-read style, and I can say his writing has improved over time. His explanations of example firewall rule sets are clearer than I recall from the second edition.

I've used PF for my home firewall for many years and plan to build a new firewall appliance (using an APU1D4), which will

also use PF. While some people might ask why not use Linux, PF has the advantages of a single rule set for both IPv4 and IPv6, the BSD license, and not having the same IP stack as the majority of servers and home routers. Diversity is good for security, and PF has a nicer syntax than Linux IP tables. Hansteen does a great job of explaining all you can do with PF.

Alan Turing: The Enigma

Andrew Hodges

Princeton Press, 1983; 736 pages

ISBN 978-0-691-16472-4

Reviewed by Rik Farrow

I got a surprise in the mail while recovering from working at LISA14: a new, paperback edition of the biography of Alan Turing. I've never read biographies, but I found myself with some time on my hands when I couldn't read anything deeply technical, while also being curious about Alan Turing.

Hodges spends the first hundred pages moving Turing to the point of creating his seminal research paper, "On Computable Numbers." Just at that point, Hodges appeared to take a tremendous detour, by describing in detail some points of mathematical philosophy that Turing found important around 1938. I almost stopped reading, but after 10 pages, I discovered that Hilbert's conjectures were actually key to Turing's idea for what we today call a Turing machine.

Hodges actually adds a lot of context about the events surrounding Turing's short professional life. You've likely heard about Bletchley Park, bombs, and the German Enigma, and how important cracking German encrypted communications was to ending the war with Germany. Hodges makes these ideas concrete by explaining that England could not have survived without importing thousands of tons of goods each month, goods which had previously come from Europe. If U-boats had succeeded in sinking 50% of the cargo ships connecting America with England, England would not have had enough supplies to feed its people, much less continue fighting an air war and preparing to invade Europe. When decryption of German naval communications failed because of changes in the German system, U-boats had reached the level of starving out England. The tide of war hung on the success or failure of cryptanalysis.

While Turing did start out as an Oxford fellow, he quickly became interested in applied mathematics. He spent time at the Princeton Institute for Advanced Study, and he and von Neumann read each other's papers. Turing developed the idea of using tubes for memory and logic circuits, although others did the electronics design. And after the war, Turing helped to design the first real electronic computers.

Hodges doesn't skimp on the more troubling side of Turing. Turing was a homosexual in England at a time where just being

homosexual (not practicing) was illegal. Hodges deftly handles how difficult being attracted to men, and having to hide this, was for an intellectual who bristled at any untruths. Turing didn't suffer fools lightly, and this also led to many problems with military and institutional hierarchies. Hodges explains that Turing couldn't understand why anyone would ever avoid telling the truth, even while he himself spent most of his life hiding a basic truth about himself.

In the end, I found the context of Turing's story as important as the telling of his life through letters, papers, and the other paper trails that people leave behind. The "halting problem" went from some ideas I had about "undecidability" to the actual solution of Hilbert's conjecture. I learned about many of the early design decisions that have shaped the field of computer architecture, all by plowing through what I first thought were meanderings. If you want to better understand the context of computer science today, reading Hodges' book can certainly help you.

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

PRESIDENT

Brian Noble, *University of Michigan*
noble@usenix.org

VICE PRESIDENT

John Arrasjid, *EMC*
johna@usenix.org

SECRETARY

Carolyn Rowland
carolyn@usenix.org

TREASURER

Kurt Opsahl, *Electronic Frontier Foundation*

DIRECTORS

Cat Allman, *Google*

David N. Blank-Edelman, *Northeastern University*

Daniel V. Klein, *Google*

Hakim Weatherspoon, *Cornell University*

EXECUTIVE DIRECTOR

Casey Henderson
casey@usenix.org

In this issue:

- 62 **11th USENIX Symposium on Operating Systems Design and Implementation**
- 86 **2014 Conference on Timely Results in Operating Systems**
- 93 **10th Workshop on Hot Topics in System Dependability**

11th USENIX Symposium on Operating Systems Design and Implementation

October 6–8, 2014, Broomfield, CO

Summarized by Radu Banabic, Lucian Carata, Rik Farrow, Rohan Gandhi, Mainak Ghosh, Yanqin Jin, Giorgos Kappes, Yang Liu, Haonan Lu, Yu Luo, Amirsaman Memaripour, Alexander Merritt, Sankaranarayanan Pillai, Ioan Stefanovici, Alexey Tumanov, Jonas Wagner, David Williams-King, and Xu Zhao

Opening Remarks

Summarized by Rik Farrow (rik@usenix.org)

OSDI '14 started with a beautiful, sunny day outside of Boulder, Colorado. The nice weather lasted the entire conference, which was also brightened by record attendance.

The program chairs, Jason Flinn (University of Michigan) and Hank Levy (University of Washington), told the crowd that they had moved to shorter presentations, just 22 minutes, so they could fit four presentations into each session. By doing this, they raised the acceptance rate from the old range of 12–14% to 18%. There were 242 submissions, with 42 papers accepted. The PC spent one-and-a-half days in meetings in Seattle and used an External Review Committee to help reduce the workload on the PC. Each PC member reviewed 30 papers (on average), down from 45.

USENIX and OSDI '14 sponsors made it possible for 108 students to attend the conference via grants. Overall attendance was also at record levels.

The Jay Lepreau Best Paper Awards were given to the authors of three papers: “Arrakis: The Operating System Is the Control Plane” (Peter et al.), “IX: A Protected Dataplane Operating System for High Throughput and Low Latency” (Belay et al.), and “Shielding Applications from an Untrusted Cloud with Haven” (Baumann et al.). There was no Test of Time award this year.

Who Put the Kernel in My OS Conference?

Summarized by Giorgos Kappes (gkappes@cs.uoi.gr) and Jonas Wagner (jonas.wagner@epfl.ch)

Arrakis: The Operating System Is the Control Plane

Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson, University of Washington; Timothy Roscoe, ETH Zürich
Jay Lepreau Best Paper Award

Simon began his talk by explaining that traditional operating systems like Linux do not take advantage of modern hardware that supports I/O virtualization, and they impose significant overheads because the kernel mediates all data accesses.

Simon introduced Arrakis, a server OS that splits the role of the kernel into the control and the data plane. The control plane lies in the kernel and is responsible for functionalities like naming, access control, and resource limits. These functionalities are used infrequently, to set up a data plane, for example. On the other hand, the functionality of the data plane is moved into applications. Applications perform I/O processing themselves by taking advantage of hardware I/O virtualization, while protection, multiplexing, and I/O scheduling are directly performed by the hardware. The copying of data between the kernel and the user space is no longer needed. A per application dynamically linked library implements the data plane interfaces which are tailored to the application. The network data plane interface allows applications to directly talk with the hardware in order to send and receive packets. The storage data plane interface allows the applications to asynchronously read, write, and flush data into its assigned virtual storage area (VSA). The storage controllers map this virtual area to the underlying physical disks.

There is also a virtual file system (VFS) in the kernel that performs global naming. In fact, the application has the responsibility to map data onto its VSA and register names to the VFS. The storage data plane also provides two persistent data structures: a log and a queue. These allow

operations to be immediately persistent, protect data against crash failures, and reduce the operations' latency.

To evaluate Arrakis, the authors implemented it in Barrelfish OS and compared its performance with Linux. By using several typical server workloads and well-known key-value stores, they show that Arrakis significantly reduces the latency of set and get operations while increasing the write throughput 9x. Arrakis also scales better than Linux to multiple cores, because I/O stacks are local to applications and are application specialized.

John Criswell (University of Rochester) asked what would happen if the Linux kernel made the hardware devices directly available to applications. Simon replied that there is a lot of related work that does try to keep the Linux kernel intact. However, it does not provide the same performance as Arrakis, since the kernel has to be called eventually. System call batching can mitigate this, however this trades off latency for higher throughput. Geoff Kuenning (Harvey Mudd College) asked whether Redis must be running in order to mediate disk I/O through its dedicated library and what would happen if someone damaged the Redis config file preventing it from starting up. Simon answered that the idea behind the indirection interface is provided by the libIO stack in Redis's dedicated library. The stack includes a server that receives I/O operations and directs them to the config file. Aaron Carol (NICTA) first pointed out that it seems that Arrakis designates a process as a host for a collection of files, and then asked what performance implications would come with accessing these files from a different process. Simon replied that the process to which the file belongs will have faster access. Different processes need to perform IPC, which typically has some costs, but Barrelfish introduced fast IPC. Finally, Peter Desnoyers (Northeastern University) asked how Arrakis performs for very high connection rate applications, e.g., a large Web server. Simon said that not every connect operation needs a control-plane call. For example, a range of port numbers can be allocated to a server with a single control-plane call.

Decoupling Cores, Kernels, and Operating Systems

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe, ETH Zürich

Gerd motivated his work with current hardware and software trends: In an era of Turbo Boost, core fusion, and dark silicon, power management is crucial. Tomorrow's OSes need to switch between specialized cores based on their workload. They should be able to dynamically reconfigure or update themselves. All this is possible in Barrelfish/DC, a multikernel architecture where OS state has been carefully factored per CPU. This allows a separate kernel to control each CPU and to treat CPUs as plug-gable devices.

The main challenge is to cleanly and quickly shut down a core. Barrelfish/DC's solution is to move CPU state out of the way quickly, then dismantle it lazily, if needed. State is encapsulated in an "OSnode" containing capabilities that represent application and OS state, as well as a Kernel Control Block representing

hardware state. OSnodes can be moved to another CPU, where they are "parked" until dismantled or restarted.

Experiments show that Barrelfish/DC can shut down nodes very quickly (in <1 ms). Moreover, the time to do so does not depend on the system load. The Barrelfish/DC team has demonstrated that their system enables various new use cases, e.g., live updates of the OS kernel or temporarily switching to a real-time kernel for a specific task.

Chris Frost (Google) asked how interrupts interacted with moving cores. Gerd explained that Barrelfish/DC handles three types of interrupts: for timers, inter-process communication, and devices. When a device driver is moved to another core, device interrupts must be disabled on the source core before the move, and the driver must poll the device state once it is running on the destination core. Srivatsa Bhat (MIT) asked whether Barrelfish/DC's energy savings could also be achieved by the power modes in today's CPUs. Gerd answered that this is possible, but that his work goes beyond power savings to explore completely new ideas. Someone from Stanford asked about the cost of dismantling a state. Gerd explained that this depends on the application (e.g., whether it uses shared message channels) and that it was impossible to give a specific number. Malte Schwarzkopf (Cambridge) asked whether this would work on non-cache-coherent architectures. We don't know, said Gerd, because such a system has not yet been built.

Jitk: A Trustworthy In-Kernel Interpreter Infrastructure

Xi Wang, David Lazar, Nikolai Zeldovich, and Adam Chlipala, MIT CSAIL; Zachary Tatlock, University of Washington

Today's kernels execute untrusted user-provided code in several places: BSD Packet Filter (BPF) programs to filter packets or system calls, DTrace programs for profiling, etc. Xi Wang started his talk by showing how hard it is to build correct and secure interpreters for such user-provided code. He and his colleagues created Jitk, a verified JIT compiler for programs in the BPF language, to eradicate interpreter bugs and security vulnerabilities once and for all.

Jitk models both a BPF program and the CPU as state machines and comes with a proof that, whenever Jitk successfully translates a BPF program to machine code, all its state transitions correspond to transitions at the machine code level. Jitk's proof builds on the CPU model and an intermediate language, Cminor, from the CompCert project. The main proof is complemented by a proof that decoding BPF bytecode is the inverse operation of encoding it, and by a high-level specification language that simplifies the creation of BPF programs. Putting these components together, users can have confidence that the semantics of well-understood, high-level programs are exactly preserved down to the machine code level.

Jitk consists of 3510 lines of code, two thirds of them proof code. The JIT's performance is comparable to the interpreter that ships with Linux. Due to the use of optimizations from CompCert, it often generates code that is smaller.

Rich Draves (Microsoft) enquired how Jitk compares to proof-carrying code. Xi Wang answered that Jitk proves strong correctness properties, whereas proof-carrying code usually demonstrates only its memory safety. Also, Jitk's proof holds for any translated program. Malte Schwarzkopf (Cambridge) wondered about the value of Jitk's proof, given the large trusted code base. Xi Wang answered that all theorem proving techniques share this problem. The trusted code base consists of layers that build on each other, and we can gain confidence by analyzing layers in isolation and trusting that errors would not propagate. Volodymyr Kuznetsov (EPFL) asked whether proven code could be isolated from untrusted code. Xi Wang pointed to the related Reflex project (PLDI 2014) where, for example, isolation between different browser tabs has been proven.

IX: A Protected Dataplane Operating System for High Throughput and Low Latency

Adam Belay, Stanford University; George Prekas, École Polytechnique Fédérale de Lausanne (EPFL); Ana Klimovic, Samuel Grossman, and Christos Kozyrakis, Stanford University; Edouard Bugnion, École Polytechnique Fédérale de Lausanne (EPFL)

Jay Lepreau Best Paper Award

Adam started by mentioning the increasing mismatch between modern hardware and traditional operating systems. While the hardware is very fast, the OS becomes the bottleneck. This results from the complexity of the kernel and its interface, while interrupts and scheduling complicate things even further. Instead, today's datacenters require scalable API designs in order to support a large number of connections, high request rates, and low tail latency.

To achieve these goals, the authors designed IX, a data-plane OS that splits the kernel into a control plane and multiple data planes. The control plane consists of the full Linux kernel. It multiplexes and schedules resources among multiple data planes and performs configuration. Each data plane runs on dedicated cores and has direct hardware access by utilizing hardware virtualization. Additionally, IX leverages VTX virtualization extensions and Dune (OSDI '12) to isolate the control plane and the data planes as well as to divide each data plane in half. The first half includes the IX data-plane kernel and runs in the highest privilege ring (ring 0), while the other half comprises the user application and libIX and runs in the lowest privilege ring (ring 3).

libIX is a user-level library that provides a libevent-like programming model and includes new interfaces for native zero-copy read and write operations. Describing the IX design, Adam briefly presented the IX execution pipeline and mentioned its core characteristics. The IX data plane makes extensive use of adaptive batching, which is applied on every stage of the network stack. Batching is size-bounded and only used in the presence of congestion. This technique decreases latency and improves instruction cache locality, branch prediction, and prefetching, and it leads to higher packet rates. Additionally, the IX data plane runs to completion of all stages needed to receive and transmit a batch of packets, which improves data cache locality. It also

removes scheduling unpredictability and jitter, and it enables the use of polling.

The authors evaluated a prototype implementation of IX against a Linux kernel and mTCP, and showed that IX outperforms both in terms of throughput and latency. Additionally, IX achieves better core scalability. The authors also tested memcached and showed that IX reduces tail latency 2x for Linux clients and by up to 6x for IX clients. It can also process 3.6 times more requests.

Brad Karp (UCL) asked whether the technique used to achieve data cache locality affects instruction cache locality. He also asked whether integrated layer processing conflicts with the techniques used in IX. Adam answered that they didn't observe that data cache locality adversely affects instruction cache locality. If the amount of data that accumulates between processing phases fits in data cache, then the instruction cache is not a bottleneck. An upper limit on the batch size also helps. Simon Peter (University of Washington) asked how the batching used in IX affects tail latency, especially with future, faster network cards. Adam said that batch limits have no impact at low throughputs because batching is not used. But even at high throughputs, batching leads to low latency because it reduces head-of-line blocking. The next question was about the run-to-completion model. Michael Condict (NetApp) asked whether no one is listening on the NIC when the core is in the application processing stage. Adam replied that while the application performs processing, the NIC queue is not being polled. Michael also asked whether this technique can be used on servers that have high variability in processing time. Adam said that IX distinguishes between I/O and background threads. Applications could use background threads for longer-duration work. They also want to use interrupts to ensure that packets are not unnecessarily dropped. However, interrupts should only be a fallback. Steve Young (Comcast) asked whether they encountered dependencies between consecutive operations due to batching. Adam answered that this was a big issue when they designed their API, but careful API design can prevent such problems. They also use a heuristic: the batch is a set of consolidated network requests from a single flow. If one fails, they skip the other requests in the flow.

Data in the Abstract

Summarized by Yang Liu (yal036@cs.ucsd.edu) and Yanqin Jin (y7jin@cs.ucsd.edu)

Willow: A User-Programmable SSD

Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, University of California, San Diego

Steve Swanson first highlighted that Willow aims to make programmability a central feature of storage devices and to provide a more flexible interface. Then he gave a retrospective view of the evolution of storage technology, starting from slow hard disks to emerging PCIe attached SSDs backed by flash or phase change memory (PCM). Unlike conventional storage, these new SSDs promise much better performance as well as more flexibility, urging people to rethink the interface between storage soft-

ware and storage device. Previously, SSDs had been connected to the host via rigid interfaces such as SATA, SAS, etc., while SSDs have flexible internal components. Thus the challenge is to expose the programmability to application developers who want to build efficient and safe systems.

Steve presented Willow, a system that (1) provides a flexible interface that makes it easy to define new operations using the C programming language, (2) enforces file system permissions, (3) allows execution of untrusted code, and (4) provides OS bypass so that applications can invoke operations without system calls.

Willow is especially suitable for three types of applications: data-dependent logic, semantic extensions, and privileged execution. Intensive and complex data analytics is not the sweet spot for Willow's design, mainly because of the wimpy CPUs inside the SSD, limited by power and manufacturing cost.

Steve then presented Willow's system architecture. Willow, which is implemented on a BEE3 FPGA board, has similar components to a conventional SSD: It contains eight storage processor units (SPUs), each of which includes a microprocessor, an interface to an inter-SPU interconnect, and access to an array of non-volatile memory. Each SPU runs a small SPU-OS, providing basic functionality such as protection. Willow is connected with the host machine via NVMe Express (NVMe) over PCIe.

Willow allows application programmers to download SSD apps to the storage device. An SSD app has three components: a host-side user-space library, the SPU code, and an optional kernel module. Willow's interface is very different from that of a conventional SSD. Host threads and SPUs rely on a RPC mechanism to communicate with each other. The RPC mechanism is sufficiently flexible so that adding new interface is easy. There is nothing storage-centric about the RPC since SPUs and host can send RPCs in any direction, from host to storage and vice versa.

Steve also introduced the trust and protection model adopted by Willow in which a file system sets protection policy while Willow firmware enforces it. In particular, he pointed out that, thanks to hardware-written processID information in the message headers, RPCs cannot be forged.

To demonstrate the usefulness of Willow, Steve guided the audience through a case study and invited them to read the paper for further details. In the talk, he showed that by implementing moderate amount of transaction support inside Willow, some applications become easy to write, with a noticeable performance gain. He also emphasized that the programmability of Willow actually makes tweaking and tuning the system faster and more convenient.

Pankaj Mehra (SanDisk) asked whether future programmable SSD can work with the new NVMe standard, given the evolution of non-volatile memory. Steve said that they are actually doing some work to answer that question, and the simple answer is yes. One of the possible ways to do that is to go through the NVMe standard and add some extensions, such as allowing generic

calls from the device to the host, which will fit in the NVMe framework. Peter Chen (University of Michigan) asked whether Steve saw any technological trends that could reduce the need for programmable SSDs, when faster buses emerge. Steve said that he doesn't see trends in that direction because latency doesn't decrease much even though PCIe bandwidth continues to grow. Thus, it is still a problem if there is too much back-and-forth communication between the host and the SSD. In addition, the programming environment on the SSD is much simpler than that on the host, making the SSD more reliable and predictable. He said he can see a consistent trend towards pushing more capable processors on SSDs, and similar trends on GPUs and network cards as well. In his opinion, this is a broad trend.

Physical Disentanglement in a Container-Based File System

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin-Madison

Isolation is vital for both reliability and performance, and is widely used in various computer systems. Lanyue Lu pointed out current system design does not have isolation in the physical on-disk structures of file systems, resulting in poor reliability and performance. He further showed several problems caused by such physical entanglement, and then proposed a novel file system called IceFS to solve the problem.

Without isolation of the on-disk structures, logically distinct files may well use co-located metadata, thus the corruption of one file can affect another unrelated file or even lead to global errors, such as marking a file system as read-only. File systems also use bundled transactions to commit changes of multiple files, causing the performance of independent processes to be entangled.

IceFS introduces a new abstraction called cubes, which are implemented as special isolated directories in a file system. The abstraction of cubes enables applications to specify which files and directories are logically related. Multiple cubes do not share the same physical resources. Any cube does not contain references to any other cube. Lanyue showed that IceFS offers up to eight times faster localized recovery and up to 50 times higher performance. He also told the audience that IceFS can reduce downtime of virtualized systems and improve recovery efficiency of HDFS.

The design of IceFS follows three core principles: (1) no shared physical resource across cubes, (2) no access dependency (one cube will not cross-reference other cubes), and (3) no bundled transactions. IceFS uses a scheme called "transaction splitting" to disentangle transactions belonging to different cubes. Lanyue demonstrated the benefits within a VMware virtualized environment and a Hadoop distributed file system, achieving as much as orders of magnitude performance gain.

Bill Bolosky (MS Research) was curious to know how block group allocation is done in IceFS and was mainly concerned about whether IceFS really got rid of global metadata. Lanyue

said that each block group is self-defined with its own bitmap and does not share metadata with other block groups. Then Bill asked how block groups themselves are assigned and suggested that there might be some global metadata at a higher level to indicate the allocation status of each block group. Lanyue agreed with him in that for each block group they store a cube ID that needs to be examined when traversing the block groups, but such information is not shared.

Shen Yang (Penn State University) asked how IceFS handled a case when there is a link across cubes—for example, hard links. Lanyue replied that first IceFS doesn't support hard links. And IceFS can detect many other cases of cross reference. When I/O is performed at runtime, IceFS can check whether source and destination belong to the same cube. Another person thought that the practice of isolation was nice, but that performance tweaks might violate POSIX security checks under failures. Lanyue responded that they store the most strict permission at the root directory, which has to be examined to access any sub-directory. This can enforce the original protection. Yang Tang (Columbia University) suggested that ideally he would want a separate cube for each file. He was curious to know whether this would work in IceFS. If not, is it better to just have partitions for complete isolation? Lanyue replied that partitions would lead to wasted space, and if one file system on one of the partitions panics, it might lead to a global crash. Partitions cannot solve the problem of slow recovery either. Finally, in the case of HDFS, it is hard to make partitions on a local node.

Ziling Huang (NetApp) wondered what the performance would be like atop an HDD where jumping across different cubes might incur more disk seek operations. Lanyue confirmed that running Vmail and SQLite on HDD with IceFS would lead to worse performance. Although their system would also work for HDD, it would be more likely to yield better performance on faster devices such as SSDs.

Customizable and Extensible Deployment for Mobile/Cloud Applications

Irene Zhang, Adriana Szekeres, Dana Van Aken, and Isaac Ackerman, University of Washington; Steven D. Gribble, Google and University of Washington; Arvind Krishnamurthy and Henry M. Levy, University of Washington

Modern applications have to handle deploying code across different environments from mobile devices to cloud backends. Such heterogeneity requires application programmers to make numerous distributed deployment decisions, such as how to coordinate data and computation across nodes and platforms, how to hide performance limitations and failures, and how to manage different programming environments and hardware resources. In addition, application programmers have differing requirements: for example, some ask for reliable RPC, while others demand caching, etc. All of these contribute to complicating the development and deployment of applications. Irene Zhang introduced a system called Sapphire, aiming to free application developers from such complex but tedious tasks. Sapphire is a distributed programming platform, which separates application

logic from deployment code. Furthermore, it makes it easy to choose and change application deployment.

Sapphire has a hierarchical structure and three layers. The top layer is the distributed Sapphire application. The bottom layer is the deployment kernel (DK), which provides as basic functionality as possible. DK provides only best-effort communication and is not fault-tolerant. The key part of Sapphire architecture is the middle layer, which is a library of deployment managers and offers control over placement, RPC semantics, fault-tolerance, load balancing and scaling, etc.

An important entity in Sapphire is a Sapphire Object (SO). The SO abstraction is key to managing data locality, and provides a unit of distribution for deployment managers (DM). A Sapphire application is composed of one or more SOs in communication with each other using remote procedure calls (RPCs).

Each SO can optionally have an attached DM. Sapphire also provides a DM library. The programmers select a DM to manage each SO, providing features such as failure handling and data cache among many others. Thus, programmers can easily compose and extend DMs to further choose, change, and build deployment.

Kaoutar El Maghraoui (IBM Research) asked how flexible Sapphire is for programmers to specify what kind of deployment they want. In addition, programmers sometimes don't really know the correct deployment requirements for their applications. Irene replied by giving an example of how code offloading can work with the DM. The code offloading DM is adaptive, and it can measure the latency of the RPC to figure out the best place to place the application. Sapphire only asks the programmer to tell whether the piece of code is computationally intensive, and it will do the rest. In contrast, the current practice is either implementing the code twice, once for the mobile side and once for the cloud side, or using some code offloading systems to do pretty complicated code/program analysis to just figure out what portion of the code can or should be partitioned out. Sapphire gets a little bit of information from the application programmer and then does something really powerful.

Howie Huang (George Washington University), asked whether Sapphire also deals with other issues such as security, scalability, and energy consumption, which are important to mobile applications. Irene replied that they haven't looked at energy yet and encouraged the building of a DM that could take energy into account. That would require the DK to monitor energy consumption of the system; right now the DK can only provide latency information. As for privacy and security issues, Irene revealed that they are actually looking at a similar system that provides the user with improved data privacy.

Phil Bernstein (Microsoft Research) asked whether Irene could give the audience an idea of how Sapphire would scale out in a cluster environment, given that the experiment was done on a single server. He noted, in addition, that the DM is centralized

and may become a bottleneck. Irene replied that there are actually some evaluation results in the paper in which they tested scalability with multiple servers and DMs. DMs themselves are not centralized: instead, there is another rarely used but centralized service to track SOs when they move or fail.

Phil then asked about multi-player games with hundreds of thousands of gamers coming and going: Can Sapphire handle the creation and destruction rate scaling up from there? Irene thought it would definitely be a problem if this happens a lot. The other thing she imagined is that Sapphire objects are shared virtual nodes, and most of them are created on cloud servers as files, just like you would for most of the games today.

Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems

Riley Spahn and Jonathan Bell, Columbia University; Michael Lee, The University of Texas at Austin; Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser, Columbia University

Riley Spahn first used an interesting email deletion as an example to motivate this work. In the example, he showed the audience how an email app can perform unexpectedly in managing user data: Although a user may believe that the email has been deleted, it is actually not and is still sitting somewhere in persistent storage and prone to malicious manipulations. Riley pointed out that this is a prevailing problem because a number of popular Android apps suffer from this. Conventional OSes lack the ability to understand high-level user data abstraction and understand plain files instead. Such a mismatch between the OS and the user's perception of data can lead to difficulty in building data protection systems on smartphones. To address this problem, Riley and his colleagues built and proposed Pebbles.

Pebbles is a system-level tool designed to recognize and manage fine-grained user data. It does not require program change, because the authors don't expect application programmers to change their code. This may seem impossible at first glance, but they made the amazing discovery that in real life, user data on smartphones have quite good uniformity, making it feasible to detect.

Logical Data Objects (LDOs) are different from ordinary files in that they can be hierarchical and span multiple data stores: e.g., plain file, key-value store, and SQLite database. To address the aforementioned problem, they made several important assumptions. First, protection tools are trusted. Second, applications which produce LDOs will not maliciously act against Pebbles by manually obfuscating them. Finally, they limit their scope to persistent data, leaving main memory aside. Given these assumptions, they want Pebbles to be able to hide some data, audit access to data, and restrict access to some data.

Pebbles is plugged into Android and modifies Android in three ways: (1) Dalvik with TaintDroid to track dataflows and discover relationships, (2) three modified storage APIs to generate relationships between them, and (3) a new system service called Pebbles Registrar to record all the relationships and create object graphs. This graph of LDOs, or object graph, is the

most significant piece of Pebbles since it represents Pebbles' understanding of application data. They used several mechanisms to build the graph, with details presented in the paper. They also built four different applications leveraging the service provided by Pebbles. Evaluation results show that Pebbles is quite accurate in constructing LDOs without supervision. The performance overhead is relatively low, and Pebbles provides reasonably good performance to application users.

Ashvin Goel (University of Toronto) was curious about whether relations other than foreign key relations and file name relationships could be detected. Riley pointed out that basically all relations that can be found are dataflow relationships. By tracking data being written to a certain file that generates a bi-directional relationship because of data sharing, Pebbles could detect a uni-directional relationship from there based on the access. Jonas Wagner (EPFL) commented that many applications want to encrypt their storage. Riley said they didn't evaluate applications that used encryption, although several hundred use a library to encrypt their SQL storage.

My Insecurities

Summarized by Radu Banabic (radu.banabic@epfl.ch) and David Williams-King (dwk@cs.columbia.edu)

Protecting Users by Confining JavaScript with COWL

Deian Stefan and Edward Z. Yang, Stanford University; Petr Marchenko, Google; Alejandro Russo, Chalmers University of Technology; Dave Herman, Mozilla; Brad Karp, University College London; David Mazières, Stanford University

Deian Stefan began by observing that today's Web apps entrust third-party code with the user's sensitive data, leaving browsers to prevent mistreatment of that data. Basic protection is provided by Web browsers' same-origin policy (SOP), where content from different sites is separated into browsing contexts (like tabs and iframes), and scripts can only access data within their own context. But SOP has two problems: (1) it is not strict enough, since a site (or libraries like jQuery) can arbitrarily exfiltrate its data, and (2) it is not flexible enough, because third-party mashup sites are prevented from combining information from multiple source Web sites. So browsers have extended SOP with discretionary access control: The Content Security Policy (CSP) allows a page to communicate with a whitelist of sites, and Cross-Origin Resource Sharing (CORS) allows a server to whitelist sites that can access its data. However, this is still not a satisfactory solution. Taking CORS as an example, if a bank grants access to a mashup site, that site can still do anything with the data (e.g., leak it through buggy or malicious software). So the bank will be unlikely to whitelist such a site, and the mashup may instead fall back on the dangerous practice of requesting the user's bank login credentials.

Deian explained that the challenge addressed by COWL is to allow untrusted third-party code to operate on sensitive data. His motivating example is an untrusted password strength checker. Ideally, the code should be able to fetch lists of common passwords from the Internet to compare against the user's

password, but as soon as the code gains access to the real user's password, it should no longer be able to interact with the network. This is a form of mandatory access control (MAC) known as confinement, for which there exists prior work, but existing confinement systems (Hails, Jif, JSFlow) are overly complex for a Web environment. COWL's design goal is to avoid changing the JavaScript language runtime and Web security model to avoid alienating Web developers.

COWL adds confinement to the existing model of the Web in a natural way. Just as browsers enforce separate execution contexts by different origins (source domains), COWL introduces data protection through labels that specify the origins which care about the data. COWL tracks labels across contexts (iframes, workers, servers). Any context may add a new origin to its label in order to read data labeled with some origin, but then can only communicate with that origin (and cannot communicate at all if its label contains two or more origins). COWL enforces label semantics for outgoing HTTP requests as well as for communication between browser contexts.

For evaluation, Deian mentioned four applications implemented on COWL (including a mashup like mint.com which summarizes banking info). COWL itself is implemented for Firefox and Chromium (by modifying 4k lines of code in each), changing the gecko and blink layout engines, and adding parameters to communications infrastructure like `postMessage()` and `XMLHttpRequest`. In terms of performance, there is no additional overhead for sites that do not use COWL (it is only enabled the first time the COWL API is called), while the overhead for the mashup example, excluding network latency, is 16% (16 milliseconds). Deian claimed that COWL can be easily deployed, given its backwards compatibility, reuse of existing Web concepts, and implementation in real Web browsers. One limitation of COWL is that it does not deal with covert channels. In addition, apps must be partially redesigned with compartmentalization in mind (simply adding labels to sensitive variables is insufficient). Some related work includes (1) BFlow, a coarse-grained confinement system for the Web which does not handle the case where two parties mutually distrust each other, and (2) JSFlow, which does fine-grained confinement, is better suited for tightly coupled libraries, and has high overhead (100x). Deian concluded by saying that today we give up privacy for flexibility to allow apps to compute on sensitive data, but the mandatory access control provided by COWL—a natural extension of the existing Web model—allows developers to do better.

The first question was about covert channels: Couldn't information be leaked by sending labeled data to another context and having it respond with one of two messages, leaking one bit of the protected data, and couldn't this process be repeated to leak the entire data? Deian answered that the intent of COWL is to close off all overt communication channels, and while covert channels might still be possible, COWL's approach is better than the current approach where a site is given all-or-nothing access to the data through discretionary access control. Mike Freedman

(Princeton) mentioned that mandatory access control systems often have trouble with declassification, and was this ever necessary with COWL, or are browsers essentially stateless? Deian answered that a site can read its own data labeled with its own origin, and this is a sufficient form of declassification. Another attendee asked about the ramifications of defaulting to open access instead of defaulting to closed access before COWL becomes universally deployed. The answer is that a site must opt-in to COWL's mandatory access control by adding a label to some data in order to loosen mechanisms like CORS, and clients that do not support COWL would fall back on the default discretionary access control as deployed today.

Code-Pointer Integrity

Volodymyr Kuznetsov, École Polytechnique Fédérale de Lausanne (EPFL); László Szekeres, Stony Brook University; Mathias Payer, Purdue University; George Candea, École Polytechnique Fédérale de Lausanne (EPFL); R. Sekar, Stony Brook University; Dawn Song, University of California, Berkeley

Volodymyr started by explaining control-flow hijack vulnerabilities: By exploiting a memory safety error in a program, an attacker can overwrite function pointers in program memory and divert the control-flow of the program to execute any code the attacker wants. Despite this being a known problem for 50 years, it is still relevant today; there are more and more control-flow hijack vulnerability reports in the CVE database every year. Code written in high-level languages avoids this problem, but such code often requires millions of lines of C/C++ code to run (language runtimes, native libraries, etc.). There are techniques to retrofit precise memory safety in unsafe languages, but the overhead of such techniques is too high for practical deployment. The control-flow integrity technique provides control-flow hijack protection at lower overhead, but many of control-flow integrity implementations were recently shown to be bypassable.

The authors proposed Code-Pointer Integrity as a technique to eliminate control-flow hijack vulnerabilities from C/C++ programs, while still keeping the overhead low. The key insight is to only protect code pointers in the program; as these are only a minority of all the pointers in the program, the overhead due to the memory safety instrumentation for just these pointers is low.

The implementation of the technique separates memory into two regions: safe and regular memory. The isolation between the two is enforced through instruction-level isolation and type-based static analysis. Instructions that manipulate program data pointers are not allowed to change values in the safe memory region, even if compromised by an attacker. This ensures that attackers will not be able to exploit memory safety errors in order to forge new code pointers. This protection mechanism is called code-pointer separation (CPS). However, this still leaves the potential of an attack, where attackers manipulate pointers that indirectly point to code pointers (such as through a struct) and are thus able to swap valid code pointers in memory, causing a program to call a different function (only a function whose address was previously taken by the program). In order to protect against this type of attack, the authors also propose the code-pointer integrity (CPI) mechanism, which also puts in the safe

memory region all sensitive pointers, i.e., all pointers that are used to indirectly access other sensitive pointers (essentially, the transitive closure of all direct and indirect code pointers). CPI has a bit more overhead than CPS but has guaranteed protection against control-flow hijacks.

In the evaluation, the authors showed that both CPS and CPI protect against all vulnerabilities in the RIPE benchmark and that CPI has a formal proof of correctness. The authors compared CPS and CPI to existing techniques, such as CFI (both coarse- and fine-grained) and ASLR, DEP, and stack cookies, showing that CPS and CPI compare favorably to all: The proposed techniques have either lower overhead, higher guarantees, or both. CPS provides practical protection against all existing control-flow hijack attack and has an average overhead of 0.5%–1.9%, depending on the benchmark, while CPI provides guaranteed protection at an average overhead of 8.4%–10.5%, depending on the benchmark. The stack protection (a component of both CPS and CPI that protects the stack) has an average overhead of 0.03%. The authors released an open-source implementation of their techniques; the protection can be enabled using a single flag in the LLVM compiler.

Jon Howell (MSR) commented that the overhead of CPI is still relatively high, at 10%, but that the attacks against CPS cannot easily be dismissed, since ROP attacks are known. Volodymyr answered that CPS still protects against ROP attacks, since the attacker cannot manipulate the return addresses in any way, and can only change function pointers to point to a function whose address was already taken by the program (and not to any function in a library used by the program). Úlfar Erlingsson (Google) commented that the authors misrepresented prior work. He argued that there is no principled attack against fine-grained CFI and that the cited overheads were true 10 years ago, but a current implementation in GCC has an overhead of 4% (instead of 10% as was cited). Finally, Úlfar asked how the proposed technique is not affected by conversions between pointers and integers, which affected PointGuard several years ago. Volodymyr answered that the analysis handles such cases, and the authors successfully ran the tool on all SPEC benchmarks, which shows the robustness of the analysis.

The next question was about switch statements: Some compilers generate jump tables for such code; is this case handled by the tool? Volodymyr answered that compilers add bound checks for the generated jump table, and they are fully covered by the tool. David Williams-King (Columbia) asked about the 64-bit implementation of the tool, where the lack of hardware support for segmentation forced the authors to use alternative techniques. David asked whether OS or future HW support would help avoid any information leak attacks. Volodymyr answered that the authors have two mechanisms that work on 64-bit, one stronger and one faster. The faster support relies on randomization that is not vulnerable to information leaks, while the stronger approach relies on software fault isolation. Joe Ducek (HP Labs) asked how much of the performance overhead in CPI is due to the

imprecision in the analysis and how much to the actual instrumentation. Volodymyr answered that most overhead comes from handling of `char*` and `void*` pointers, which in C/C++ are universal, but `char*` is also used for strings; the tool needs to protect all occurrences of these types of pointers, which leads to the majority of the overhead.

Ironclad Apps: End-to-End Security via Automated Full-System Verification

Chris Hawblitzel, Jon Howell, and Jacob R. Lorch, Microsoft Research; Arjun Narayan, University of Pennsylvania; Bryan Parno, Microsoft Research; Danfeng Zhang, Cornell University; Brian Zill, Microsoft Research

Bryan Parno started by pointing out the very weak guarantees that users have today when submitting private data online. The only guarantees come in the form of a promise from service providers that they will use good security practices, but a single mistake in their software can lead to a data breach. In contrast, Ironclad, the approach proposed by the authors, guarantees that every low-level instruction in the service adheres to a high-level security specification. Ironclad relies on HW support to run the entire software stack in a trusted environment and on software verification to ensure that the software respects a high-level specification. The hardest part is software verification of complex software; in Ironclad, the authors managed to go a step beyond the verification of a kernel (the seL4 work), by verifying an entire software stack with a reasonable amount of effort (without trusting OS, drivers, compiler, runtime, libraries, etc.). To allow this, the authors had to abandon the verification of existing code and rely on developers to specifically write their software with verification in mind.

First, developers write a trusted high-level specification for the application. Then they write an untrusted implementation of the application. Both the specification and implementation are written in a high-level language called Dafny. The implementation looks like an imperative program, except that it has annotations, such as contracts and invariants. The specification is translated by Ironclad to a low-level specification that handles the low-level details of the hardware on which the application will run. Similarly, the implementation is compiled to a low-level assembly language, where both code and annotations handle registers, instead of high-level variables; the compiler can also insert some additional invariants. Finally, a verifier checks whether the low-level implementation matches the low-level specification, and then the app can be stripped of annotations and assembled into an executable.

Bryan gave a live demo of how the system works. The demo showed that Ironclad provides constant, rich feedback to the developer, significantly simplifying the process of writing verifiable code.

The system relies on accurate specifications of the low-level behavior of the hardware that is used. Writing such specifications seems like a daunting task; the Intel manual, for instance, has 3439 pages. The authors bypassed this issue by only specifying a small subset of the instructions, and enforcing the rule that

the system only use those instructions. Similarly, the authors developed specifications for the OS functionality and for libraries. In order to prevent information leaks, Ironclad uses a Declassifier component, which checks whether any data output by the implementation to the outside world (e.g., over the network) would also be output by the abstract representation of the app in its current state.

When discussing the evaluation of Ironclad, Bryan first pointed out that the development overhead for writing a verifiable system wasn't too bad: The authors wrote 4.8 lines of "proof hints" for every 1 line of code in the system. Moreover, part of this effort will be amortized further over time, since a bulk of the proof hints were in reusable components, like the math and standard library. Even as it is now, the ratio of 4.8:1 is significantly better than previously reported ratios of 25:1. In terms of total number of lines of code, the trusted specification of the system has ~3,500 lines of code, split just about evenly between hardware and software, and ~7,000 lines of code of high-level implementation. The latter get compiled automatically to over 41,000 assembly instructions, which means that the ratio of low-level code to high-level spec is 23:1. In terms of performance, initial versions of the implementation were much slower than their non-verifiable counterparts but are amenable to significant manual optimizations; in the case of SHA-256 OpenSSL, the verifiable application is within 30% of the performance of the native, unsafe OpenSSL. The code, specification, and tools of Ironclad will be made available online.

One attendee asked the presenter to say a few words on concurrency. Bryan answered that the authors are currently working on a single processor model; some colleagues are working on multicore, but the state of the art in verification for multicore processors is way behind that for single-threaded programs. The next question was whether the authors have any experience with more complex data structures, such as doubly linked lists. The answer was that the data structures used so far were fairly simple, and most of the time was spent on number-theoretic proofs. Someone from Stanford asked whether the verification could be extended to handle timing-based attacks. Bryan answered that they have not looked into that yet, but there are other groups that are considering timing: for example, the seL4 project. Gernot Heiser (NICTA and UNSW) commented that the entire verification relies on the correctness of the specification and that the authors' approach to ensure correctness is a traditional top-down approach, which is known not to work for real software. He then asked how it is possible to ensure specification correctness for software that uses more modern software development approaches. Bryan answered that there is always a human aspect in the process and that the authors found spec reviews particularly useful. Also, one can pick a particular property of interest, such as information flow, and prove that property against the specification. Finally, Kent Williams-King (University of British Columbia) asked what happens if an annotation is wrong. Bryan replied that the annotations are only used by the verifier as

hints. If an annotation is invalid, the verifier will complain to the user and discard the annotation for the rest of the proof.

SHILL: A Secure Shell Scripting Language

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong, Harvard University

Scott Moore opened by describing how difficult it can be to figure out what a shell script does. Such comprehension requires fully reading the script's code and understanding its execution environment. Yet, going against the Principle of Least Privilege, every script runs with far broader privileges than it needs to do its job. This has security implications since, as in the case of the Shellshock vulnerability, every instance of bash that runs a malicious script can open network connections and execute with the full privileges of its invoking user. Scott then asked two questions: How can the authority of scripts be limited, and how can the authority necessary for a script's operation be determined? To answer these questions, Scott presented Shill, a scripting language where every script comes with its own declarative security policy. In addition to enforcing runtime restrictions, the policy can also be examined by the user to decide whether the script seems safe. Shill scripts can recursively call other Shill scripts if the policy allows, or invoke native executables, which are run inside a sandbox to ensure end-to-end enforcement of the original security policy.

Besides sandboxing, Shill's implementation relies on capabilities, which make the script's authority explicit rather than an artifact of its environment, and contracts, which are the declarations describing how capabilities can be used. Capabilities are an unforgeable token of authority, the possession of which grants the right to perform some action (like keys open doors). This contrasts with existing mandatory access control mechanisms, like UNIX file permissions, which are a property of the environment. There has been a great deal of related work on capabilities. In Shill, functions take capabilities as parameters: files, directories, pipes, sockets are each represented as a capability. Operations like opening or reading a file require privileges on the capability (and "opening" a file in a directory returns a derived capability for the file). All resources are represented as capabilities, and the only capabilities a script has are the ones passed in, making it easy to reason about a script's effects; this is termed "capability safety."

Software contracts in general essentially specify pre- and post-conditions that can be executed to verify that a program runs as it should. In Shill, every function has a grammatically compatible specification written before it; the Shill interpreter checks for contract violations at runtime, and if any are found, terminates the script. The contracts may list the privileges required by the script for each capability (e.g., list files in directory). A callee may assume it has these privileges; the caller can use the privileges to reason about the possible side effects of the call. This aids in reasoning about composing scripts together. This reasoning can extend to any native binaries the Shill script invokes, because Shill sandboxes binaries (without modifying them) to

continue checking for contract violations. In the presentation, Scott gave an example of a sandboxed program opening a file in another directory, and all the privilege checks that occur. Since running most programs requires a large number of (file) capabilities, Shill supports capability wallets, which are composite capabilities necessary to perform common tasks. Bootstrapping—providing the capabilities necessary to run the original script—is achieved with ambient scripts, which are limited and can only create capabilities to pass to Shill scripts.

Shill is implemented in a capability-safe subset of the Racket programming language, and Shill's sandbox is implemented in TrustedBSD's MAC framework. Scott described several case studies for Shill, including an Emacs installer script (which can only create/remove files in certain directories), a modified Apache server (which can only read its configuration and content directories), find-and-grep, and a grading script. In the last case, a TA is tasked with grading programming assignments from many different students using a test suite. The TA might create a script to automate the compiling, running, and test verification process; but it is difficult to ensure that each student's assignment doesn't corrupt other files, leak the test suite, or interact with other students' code. When the script is written in Shill, the security policy allows the TA to reason that the students' assignments will behave themselves, even if the TA's own script is buggy. In terms of performance, Shill generally has less than 20% overhead on these four examples, except find-and-grep which may spawn many sandboxes, leading to 6x overhead. The overhead is proportional to the security guarantees. In summary, Shill allows the Principle of Least Privilege to be applied to scripting, using a combination of capabilities, contracts, and sandboxing. The code and a VM with Shill are available online.

Xu Zhao (University of Toronto) asked about the motivating example of downloading a large untrusted script from the Internet, because such a script might have a very complex security policy. Scott's answer was that with existing scripts, the whole script must be scrutinized along with its execution environment, whereas the security policy provides a single place to focus one's attention. A student-attendee from Columbia University asked why use capabilities instead of access control, and how does Shill compare with SELinux. Scott answered that SELinux creates system-wide policies, while Shill allows more fine-grained control, and Shill turns the question about whether a script is safe to run into a local decision instead of a question about the environment.

Brian Ford (Yale) asked about confused deputy attacks on the sandbox, where an incorrect capability is used to gain access to a resource. Scott answered that to mitigate such attacks, components could be rewritten in Shill to leverage its security checks. Stefan Bucur (EPFL) asked about the development time overhead for programmers writing Shill (since many scripting languages are used for quick prototyping). Scott answered that it is similar to writing in Python instead of in bash; one has to think in terms of data types instead of paths. But it is possible to

start with broad, permissive security policies and refine them later. Someone from UC San Diego asked whether the authors had applied security policies to existing scripts to see how many misbehave. Scott replied that the closest they got was translating some of their own bash scripts into Shill. Someone from CU Boulder asked about enforcing security policies across multiple machines through ssh. Scott explained that Shill will not make guarantees about the remote machine, but it will control whether a script is allowed to create ssh connections in the first place.

Variety Pack

GPUnet: Networking Abstractions for GPU Programs

Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, and Emmett Witchel, The University of Texas at Austin; Amir Wated and Mark Silberstein, Technion—Israel Institute of Technology

Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Mark Silberstein presented GPUnet, a socket API for GPGPU applications. The aim is to target the development efforts of server-based GPGPU applications: Due to both their complex designs and the burden of manually managing data flow between the GPGPU, network, and host memory, such systems require more time and effort to develop and make efficient. Costs arise from the coordination of data movement between the GPGPU, DRAM, and NIC using only low-level interfaces and abstractions (e.g., CUDA). As is the nature of modern GPGPUs, a host CPU is required to hand off data or work between the GPGPU and other devices, adding complexity and overhead. Mark argues that such a CPU-centric design is not optimal, and that GPGPUs should be viewed as peer processors instead of as co-processors. The lack of I/O abstractions for GPGPU applications, however, makes this challenging.

To avoid costs of data movement and synchronization complexity placed on developers, GPUnet provides a socket API for GPGPU-native applications. Additional contributions include a variety of optimizations enabling GPGPU-based network servers to efficiently manipulate network traffic, and the development and evaluation of three workloads using GPUnet: a face verification server, a GPGPU-based MapReduce application, and a matrix product workload.

Underlying their interface are two example designs that evaluate where one can place the execution of the network stack. The first resembles modern approaches where the CPU processes network packets, utilizing the GPGPU for accelerated parallel processing of received data, and scheduling data movements between the NIC and the GPGPU. A second design exports the network stack to execute natively on the GPGPU itself, where most of the effort involved was in porting the CPU code to the GPGPU. The latter design removes CPU-GPGPU memory copies, as the host CPU can schedule peer-to-peer DMA transfers using NVIDIA's GPUDirect. Their implementation provides two libraries exporting a socket API, one for CPU-based code and the other for GPGPU-based codes.

The design of an in-GPGPU-memory MapReduce application uses GPUfs (prior work) to load image and other data from the host disk to GPGPU-resident memory buffers. Experiments were performed using both one and four GPGPUs each, showing speedups of 3.5x for a K-means workload, and 2.9x for a word-count workload. The latter has a lower performance gain due to its I/O-intensive nature, leaving little room for opportunities from the use of GPUnet. A second application was implemented representing a face verification workload. One machine hosts the client code, a second an instance of memcached, and a third the server implementation. Three implementations for the server were compared: CPU-only, CUDA, and GPUnet. Results show the GPUnet implementation to provide less variable and overall lower response latencies.

Their code has been published on GitHub at <https://github.com/ut-osa/gpunet>.

Rodrigo Fonseca (Brown University) questioned the use of GPUnet over something like RDMA, which already bypasses the socket interface. Mark responded that there is a usability vs. performance tradeoff, and that they are currently working on a socket-compatible zero-copy API. Pramod Bhatotia (Max Planck Institute for Software Systems) asked for a comparison of this work with the use of GPUDirect. Mark clarified that their work leverages GPUDirect for registering GPGPU memory directly with the InfiniBand network device. Another attendee asked for thoughts on obtaining speedup for the more general case of applications, as opposed to the event-driven, asynchronous workload designs presented. It is a more philosophical discussion, Mark responded; GPUnet gives the freedom to choose where to host the network API. If a workload is parallel and suited for execution on a GPGPU, then you are likely to achieve speedups.

The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services

Michael Chow, University of Michigan; David Meisner, Facebook, Inc.; Jason Flinn, University of Michigan; Daniel Peek, Facebook, Inc.; Thomas F. Wenisch, University of Michigan

Summarized by Yu Luo (jack.luo@mail.utoronto.ca)

Michael Chow presented a technique (the Mystery Machine) to scale familiar performance analysis techniques such as critical path analysis on complex Web sites such as Facebook. The approach to deriving a causal relationship between different components is divided into four steps: identifying segments, inferring a causal model, analyzing individual requests, and aggregating results. Identifying segments refers to coming out with a waterfall diagram of segments executed in a request. Existing logs are aggregated and components are identified. To infer a causal model from the waterfall diagram, we can automatically analyze a large number of traces to find relationships such as happens-before, mutual exclusion, and pipelines. Through the generated causal model, we then apply it to the individual traces. The final step aggregates the results and builds up statistics about the end-to-end system. An earlier method to derive a causal model is through instrumentation of the entire

system. Another method is to have every engineer on the team draw up a model of the entire system. Both methods do not scale well. The Mystery Machine applies the four-step approach to provide a scalable performance analysis on large complex Web sites such as Facebook, which allows it to do daily performance refinements.

Greg Hill (Stanford) asked how to deal with clock drifts between machines. Michael answered that there are techniques outlined in the paper. A short answer is that the Mystery Machine assumes the round-trip time (RTT) is symmetric between client and server. It then looks at repeated requests and calculates the clock drift. Someone from the University of Chicago asked how to deal with request failure. Michael answered that this is a natural variation in the server processing time and is taken into consideration. Ryan (University of San Diego) asked how to deal with inaccurate lower level logging messages. Michael replied that we cannot do anything about it.

End-to-End Performance Isolation through Virtual Datacenters

Sebastian Angel, The University of Texas at Austin; Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska, Microsoft Research
Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Sebastian Angel began by stating that tenants are moving away from enterprise datacenters into multi-tenanted cloud datacenters. The key benefits are cost saving and elasticity. Datacenters offer services implemented via appliances. Because these appliances are shared among the tenants, the performance is degraded and unpredictable at times. Aggressive tenants tend to consume a majority of resources. Tenants should get end-to-end guarantees regardless of any bottleneck resources. To achieve the end-to-end guarantees, Sebastian introduced the virtual datacenter (VDC) abstraction and associated virtual capacities. The VDC abstraction is implemented by an architecture called Pulsar. It requires no modification to appliances, switches, guest OSes, and applications. Pulsar can allocate virtual resources based on policies from both tenants and provider. Tenants may specify how VDC resources are divided to VMs. The provider may specify the distribution of resources to maximize profit or to attain fair distribution. The VDC data plane overhead is 2% (15% for small requests) and 256 bytes/sec in the control plane for each VM.

The first questioner (Stanford University) asked how often the application needs to reevaluate relationships between tokens and requests. Sebastian answered that cost functions are fixed. The same person then asked if VDC offers latency guarantees. Sebastian answered that VDC does not offer latency guarantees. Tim Wood (George Washington University) asked how to map performance to tokens. Sebastian answered that tenants can use research tools to take high-level requirements, such as job completion time, and map them to tokens. Henry (Stanford University) asked when there are a lot of short-lived applications, have they considered what would happen during the dip in performance during the initial Pulsar capacity estimation phase?

Sebastian answered that the estimation phase time is configurable and thus the dip in performance can be adjusted.

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, University of Toronto
Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Ding Yuan presented a study of the most common failures found within distributed services. The key finding was that failures are a result of a complex sequence of events between a small number of machines, mostly attributed to incorrect error handling within the source code.

The group developed a tool called Aspirator, a simple rule-based static checker, that uncovered 143 confirmed bugs in systems evaluated by the study, such as HBase, HDFS, and Redis. Ding illustrated the severity of failures in current systems; some can even be catastrophic. As an example, when Amazon's AWS experienced a brief outage, it took down many unrelated Web sites during that time. Why would such services fail like this? They studied end-to-end failure propagation sequences between elements composing such services, as prior studies had examined failures of elements individually in isolation, such as correlations between failures and configuration file mistakes. This study examined interactions between elements themselves.

The findings were very interesting. Starting with 198 user-reported failures, they studied the discussions between developers and users via the mailing lists and were able to reproduce 73. Of these, 48 were catastrophic—those which led to system instability or downtime affecting at least a large majority of users; 92% of the catastrophic failures were a result of largely trivial bugs, such as incorrect error handling of non-fatal errors in Java code (try/catch blocks). An example of such was given in the use of ZooKeeper, where a race condition caused two events to signal the removal of a ZooKeeper node. One attempt resulted in an error, the handling of which led to an abort. Seventy-seven percent of failures required more than one input event, leading to very complex scenarios mostly found on long-running systems. Other interesting results uncovered included: 88% of failures were due to the specific order of events in multi-event sequences, and 26% were non-deterministic (the ZooKeeper example falls into this classification).

By focusing on the 92% of failures that were a result of bad error handling, Ding said, they built a static checker to detect such bugs by feeding it the Java bytecode. Three rules were employed by the checker to signal a possible bug: the error handler was empty, aborted, or had some comment such as “TODO” or “FIXME”. The checker was run on nine real-world systems and uncovered a multitude of bugs. Developers gave mixed feedback after having been notified of the group's findings: 17 patches were rejected, but 143 confirmed fixes were adopted. Responses included, “Nobody would have looked at such a hidden feature”

and “I fail to see why every exception should be handled.” The reason for mixed responses is due to prioritization of developer responsibilities, among other things, such as developers thinking errors will not happen, evolving code, and bad judgment.

Many audience members praised this work prior to asking their questions. A researcher from IBM Research asked whether the problem of ignoring exceptions could be solved through static analysis. Ding asked in return whether she thought she meant to remove the burden of handling all exceptions from the developers. She clarified to mean just the gaps should be filled. Ding responded with skepticism. Error handling is messy, as seen from the examples. Doing so is definitely burdensome for developers, but automating this may lead to even more catastrophic failures. A researcher at NC State asked Ding to explain the 8% of the bugs that were not caused by mishandled exceptions. Ding replied that this 8% represented silent errors not seen by developers. An example in Redis was a failure that resulted from too many file descriptors, a scenario not verified by developers. The researcher followed up with a thought that error masking at the lower levels in the software stack may affect this. Ding suggested that it is hard to suggest more generally which layer in the stack is responsible for handling any given error, except to suggest that an error should be returned to the layer that is most apt to deal with it. He said he was unsure if either silent handling or masking are appropriate techniques in the general case. It might be best to just return the error to the developers, but it is a profound question to which he really can't provide a definite answer. Finally, a researcher (John) asked Ding to compare the tool against something like FindBugs. Ding replied that FindBugs has checks for around 400 scenarios but not for the specific patterns they looked for in this study.

Posters I

Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Unit Testing Framework for Operating System Kernels

Maxwell Walter, Sven Karlsson, Technical University of Denmark

New operating system kernels need to be tested as they are being developed, before hardware becomes available, and across multiple hardware setups. To make this process more streamlined and manageable for kernel developers, Maxwell's proposed system leverages system virtualization with a new testing API. A kernel is booted inside a virtualized environment using QEMU and is presented as virtual hardware configurations, or devices configured as pass-through, e.g., using IOMMUs. A kernel testing API they develop enables a client to use their framework to specify means for creating and executing tests. Capturing state for post-analysis is accomplished via Virtual Machine Inspection (VMI), enabling users to inspect kernel and virtual machine state to locate sources of bugs or race conditions. One limitation is that the virtual implementation of devices and hardware presented to kernels within QEMU behave ideally, unlike real hardware.

Head in the Cloud

Summarized by Alexey Tumanov (atumanov@cmu.edu) and Rohan Gandhi (gandhir@purdue.edu)

Shielding Applications from an Untrusted Cloud with Haven

Andrew Baumann, Marcus Peinado, and Galen Hunt, Microsoft Research
Jay Lepreau Best Paper Award

Andrew Baumann introduced Haven, a prototype to provide shielded execution of an application on an untrusted OS and hardware (except CPU). The motivation for Haven stems from limitations of existing cloud platforms to support trusted computing, where the cloud user has to trust the cloud provider's software, including privileged software (the hypervisor, firmware, etc.), the management stack, which could be malicious, administrative personnel or support staff who can potentially have access to the cloud user's data, and law enforcement. The existing alternatives to this problem are also severely limited for general purpose computing, e.g., hardware security modules (HSMs) that are expensive and have a limited set of APIs.

Haven aims to provide similar guarantees as guarantees provided by "secure collocation of data," where the only way for an outsider to access the data is through the network and not through other hardware and software. In this environment, the cloud provider only provides resources and untrusted I/O channels. Haven ensures confidentiality and integrity of the unmodified application throughout its execution.

Haven makes several key contributions. First, it provides shielded execution using Intel's SGX that offers a process with a secure address space called an "enclave." Intel SGX protects the execution of code in the enclave from malicious code and hardware. SGX was introduced for protecting execution of a small part of the code and not large unmodified applications. Haven extends the original intent of SGX to shield entire applications, which requires Haven to address numerous challenges, including dynamic memory allocation and exception handling. Second, Haven protects the applications from Iago attacks where even the OS can be malicious and the syscalls can provide incorrect results. Haven uses an in-enclave library to address this challenge. Third, Haven presents the limitations of the SGX as well as a small set of suggestions to improve shielded execution.

Haven was evaluated based on the functional emulator, as the authors don't have any access to the current SGX implementation. The authors constructed a model for SGX performance considering TLB flush latencies, variable delay in instruction executions, and a penalty for accessing encrypted memory. In the pessimistic case, Haven can slow down execution by 1.5x to 3x.

John Stalworth asked whether they used a public key for attestation and who owns the key. Andrew replied that Intel provides attestation through a group signature scheme and suggested an Intel workshop paper for details. The owner of the key will be the processor manufacturer (Intel). Nicky Dunworth (UIUC) asked about the programming model and about legacy applications.

Andrew again redirected the questioner to the Intel workshop paper with a remark that they still need to support legacy applications due to their large number. Another questioner wondered about Haven's limitations, especially about the memory size/swapping. Andrew said that the size of the memory is fixed, and paging is supported in hardware. John Griswald (University of Rochester) asked about the impact of the cloud provider disabling the SGX. Andrew responded that applications can still run but the attestation will fail.

Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing

Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, and Jingren Zhou, Microsoft; Zhengping Qian, Ming Wu, and Lidong Zhou, Microsoft Research

Eric Boutin introduced Apollo as a cluster scheduler deployed at Microsoft to schedule cloud-scale big data jobs. These jobs are typically specified in a higher-level (SCOPE) SQL-like language that compiles to a DAG of tasks. Apollo sets the goal to minimize job latency while maximizing cluster utilization given the challenges of scale and heterogeneous workloads. Specifically, the target scale of this system is 170k tasks over 20k servers with 100k scheduling requests per second.

First, Apollo adopts a distributed coordinated architecture approach, with one independent scheduler per job making independent decisions but with access to global cluster information, aggregated and provided by the Resource Monitor (RM). Second, Apollo introduces the abstraction of the wait-time matrix, which captures estimated wait time for a given <CPU, Mem> resource request. It masks the underlying heterogeneity of the hardware by grouping servers of the same capacity (e.g., quad-core, 8 GB nodes) and reporting the estimated wait time to acquire that type of server. The wait-time matrix allows Apollo to minimize the total task completion time, simultaneously considering both the queueing delay and the effect of a given node on execution runtime of the scheduled task. The authors refer to this as estimation-based scheduling.

Apollo has the ability to reevaluate prior scheduling decisions. When the scheduler's updated placement decision differs from the one previously made, or there's a conflict, Apollo issues a duplicate task to the more desired server. Lastly, Apollo uses opportunistic scheduling to allow currently running jobs to allocate additional tasks above their allowed quota. This helps Apollo reach their desired goal of maximizing cluster utilization.

Christos Kozyrakis (Stanford) asked about interference of tasks co-located on the same node, sharing cache, disk/flash bandwidth, etc. Christos was specifically interested in whether the problem of interference was measured. The answer was no; Apollo depends on robust local resource management to isolate performance, like JVM and Linux Containers. Henry (Temple University) asked whether the authors considered utilization for other resource types, like disk I/O or memory. Eric replied that the CPU was the primary resource they optimized for in their particular environment. Memory utilization was recognized as important, but no numbers for memory utilization were

published in the paper. Lastly, disk I/O was also recognized as important for good performance, but Christos repeated that Apollo, relied on the local node performance isolation mechanisms “to do their job.”

Malte Schwarzkopf (University of Cambridge) pointed out that the formulation used for the wait-time and runtime assumed batch jobs. Related work (Omega) looked at placing other types of jobs, such as service tasks. The question was whether Apollo had support for that or could be extended to support it. At a first approximation, Eric argued that long-running services could be modeled as infinite batch jobs. It would simply block out the corresponding rows and columns of the wait-time matrix. Malte’s follow-up concern was that it would not lead to a good schedule, as there are no tradeoffs to be made if tasks are assumed to be running forever. Additionally, the quality of service task placement also varies. Eric responded that Apollo was targeted at cloud-scale big data analytics, with the architecture generally supportive of other schedulers, such as a service scheduler.

Vijay (Twitter) asked about dealing with discrepancies in load across clusters, wondering whether Apollo could make considerations across multiple cluster cells, based on load. The answer was no, as Apollo’s workload is such that data locality dominates the decision about which cluster the job would run in.

The Power of Choice in Data-Aware Cluster Scheduling

Shivaram Venkataraman and Aurojit Panda, University of California, Berkeley; Ganesh Ananthanarayanan, Microsoft Research; Michael J. Franklin and Ion Stoica, University of California, Berkeley

Shivaram Venkataraman stated that the context for this work is that the volume of data and jobs that consume it grows, while job latency is expected to drop, down to the near-interactive range of seconds. One specific feature of jobs exploited by this work is the ability to work with a subset or a sample of the data instead of the whole data set. Applications that exhibit these properties include approximate query processing and ML algorithms. The key insight is that the scheduler should be able to leverage the choices afforded by the combinatorial number of K -samples out of N units of data. In this work, the authors’ goal is to build a scheduler that’s choice-aware. Shivaram presented KMN Scheduler, which is built to leverage the choices that result from choosing subsets of data blocks to operate on.

Making systems aware of application semantics, finding a way to express application specifics to the scheduler, in other words, is shown to be highly beneficial for this narrow class of applications that benefit from operating on the subset of their data. The authors explore ways to propagate the choices available to the applications to the scheduler and thus leverage the flexibility that is present. Using such a system was shown to improve locality and also balance network transfer, with evaluation evidence that it benefits this emerging class of applications.

Bill Bolosky (Microsoft Research) pointed out that statistical theorems about sampling assume random sampling. Data locality makes that pseudo-random, but particularly concerning is

the fact that the data-dependency in the execution time of map tasks coupled with picking first map finishers could really skew results. The authors found no discernible difference between picking the first finishers versus waiting for all map tasks to finish. The data skew is likely the one that exhibits the most amount of determinism. System effects on stragglers are otherwise mostly non-deterministic, as also supported by prior literature. Non-determinism allegedly helps the authors get away from the issues that arise as a result of map task duration skew.

Callas (VMware) was happy to see a DB talk at OSDI. He had a follow-up question about randomness and sampling based on random distributions. The issue is that depending on the partitioning of data across racks (range partitioning) may also skew results. Shivaram pointed out that KMN only counts the number of blocks that are coming from each rack, not their size, which was left for future work. Being agnostic to size gives KMN the advantage that partition size differences do not bear as much of an effect.

Malte Schwarzkopf (University of Cambridge) pointed out that Quincy may yield better results. A major fraction of this work is about trading off locality and minimizing cross-rack transfers. Quincy is actually very closely related to this work, taking the approach of modeling this problem as an optimization problem, instead of using heuristics. Quincy does not support “ n choose k ” in the general case, because it does not support combinatorial constraints. In the KMN scheduler, however, getting $m > k$ out of n is allowed, and Quincy does have the ability to model this, by carefully assigning increasing costs. The question is how well does Quincy compare to the KMN scheduler, given that Quincy’s optimization approach may actually yield better results than KMN Scheduler? Shivaram admitted that the authors haven’t tried to apply optimization on top of their solution. No comparison with Quincy was made. The starting point was the “ n choose k ” property, which was subsequently relaxed to help with cross-rack transfers. This discussion was taken offline.

Heading Off Correlated Failures through Independence-as-a-Service

Ennan Zhai, Yale University; Ruichuan Chen, Bell Labs and Alcatel-Lucent; David Isaac Wolinsky and Bryan Ford, Yale University

Ennan presented work on providing independence-as-a-service. Previously, reliability against failures was provided through redundancy, but seemingly independent systems may share deep and hidden dependencies that may lead to correlated failures. Ennan highlighted multiple examples of correlated failures, e.g., racks connected to the same aggregation switch, the EBS glitch that brought multiple physical machines down in Amazon, the correlated failures across Amazon and Microsoft clouds due to lightning.

The focus of this work is to prevent unexpected co-related failures before they happen. The authors propose INDaaS (independence-as-a-service) that tells which redundancy configurations are most independent. INDaaS calculates an independence score based on the notion of the data-sources (servers/VMs or even

cloud providers) that hold the copy of the data. INDaaS automatically collects dependency data from different data-sources and produces a dependency representation to evaluate the independence of the redundancy configuration. However, collecting data-sources might not be possible across cloud service providers, as cloud service providers won't share all the details about the sources in their infrastructure. To address this, INDaaS offers a privacy preserving approach to privately calculate independence. Each data source can privately calculate the independence score and relay it to the INDaaS.

INDaaS faces several non-trivial challenges including: (1) how to collect dependency data, (2) how to represent collected data, (3) how to efficiently audit the data to calculate independence score, and (4) how to do it privately, when dependency data cannot be obtained. INDaaS calculates dependency data using existing hooks provided by the cloud provider (details in paper). Dependency representation uses fault graphs that consist of DAG and logic gates (AND/OR gates). To efficiently audit the data, INDaaS provides two algorithms with the tradeoff between cost and accuracy. Lastly, INDaaS privately calculates the independence score using Jaccard similarity.

INDaaS was evaluated using (1) case studies based on its deployment at Yale (detailed in paper); (2) efficiency and accuracy tradeoffs between the two algorithms (minimum fault set, failure sampling) using the fat tree network topology, in which the failure sampling algorithm detected important fault sets in 1 million sampling runs in 200 minutes; and (3) network and computation overhead compared to the KS protocol (details in paper).

Mark Lillibridge (HP Labs) asked about other (ill) uses of the system. Amazon can use INDaaS to see what infrastructure and dependencies they have in common with Microsoft. Ennan answered that might be hard since INDaaS only provides the independence score and not any specifics. Someone from Columbia University asked about handling the failure probability of each individual component. Ennan said that in practice it is hard to get correct probability numbers. INDaaS relies on the numbers provided by the cloud service provider. Another questioner wondered how to find configuration that achieves 99.9999% uptime. Ennan noted that it might not be straightforward because INDaaS ranks different configurations based on their independence score.

Storage Runs Hot and Cold

Summarized by Amirsaman Memaripour (amemarip@eng.ucsd.edu) and Haonan Lu (haonanlu@usc.edu)

Characterizing Storage Workloads with Counter Stacks

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas Harvey, and Andrew Warfield (Coho Data)

Jake started his presentation with a demonstration of storage hierarchy and how it has changed over the years, with the aim of better performance for lower cost. We have been adding more layers to this hierarchy in order to bridge the latency gap between different technologies, making provisioning of storage

systems challenging and not-optimized. A major problem in this area is data placement that requires knowledge about future accesses, which is speculated based on previous data access patterns. An example of such speculation techniques is LRU, which tries to move least recently accessed data to lower layers of the storage hierarchy. However, it does not always result in optimum placement decisions. Additionally, its accuracy is time-variant and varies from application to application. Jake then posed the question, "Can we do reuse-distance computing for each request in a more efficient way?" and answered with a "Yes."

He proposed a new data structure, called Counter Stacks, and a set of matrix calculations that will be applied to this structure to compute reuse-distance. The basic idea is to have a good approximation of miss ratio curves with less memory usage. The initial version of the algorithm was quite expensive, so he went through a set of optimizations, including down sampling, pruning, and approximate counting, to make the algorithm run online. He also introduced a method to reduce the memory usage of the algorithm, making it possible to keep traces of three terabytes of memory within a 80 megabytes region. He concluded his talk by going over a list of applications and pointing out that the accuracy of their algorithm is related to the shape of the miss ratio curve.

Michael Condit (NetApp) asked about their memory usage and how they can perform computation while only keeping a portion of the access matrix. Jake pointed out that the algorithm only requires non-zero elements to do the required computations. Scott Kaplan (Amherst) asked about how the proposed method compares to the previous work in this area. Jake pointed out that those methods only maintain workload histories over a short period, not for the entire workload. Consequently, those methods will not be applicable to their cases. Tim Wood (George Washington University) suggested using the elbows in the miss ratio curve to improve the effectiveness of the proposed algorithm.

Pelican: A Building Block for Exa-Scale Cloud Data Storage

Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, and Sergey Legtchenko, Microsoft Research; Aaron Ogus, Microsoft; Eric Peterson and Antony Rowstron, Microsoft Research

Starting his presentation with a chart comparing different storage technologies in terms of latency and cost, Sergey argued that we need to have a new storage abstraction to efficiently store cold data, which is written once and read rarely. Currently, we are storing cold data on hard disk drives, where we are accustomed to storing warm data. Pelican is going to fill this gap by providing better performance than tape but with similar cost. Their prototype provides 5+ PB of storage connected to two servers with no top-rack switch.

Due to power, cooling, bandwidth, and vibration constraints, only 8% of disks can be active at any given time. In this architecture, disks are grouped into conflict domains where only one domain can be active at a time. Having these conflict domains, they designed data placement mechanisms to maximize concurrency while keeping conflict probability minimized at the same

time. Applying a set of optimizations such as request batching, Pelican can provide a throughput close to having all disks active concurrently. In terms of performance, Pelican consumes 70% less power on average compared to the case that all disks are active concurrently. However, it will add 14.2 seconds overhead for accessing the first byte of an inactive group.

Someone pointed out that there are more recent works that they have not considered in their related works. He mentioned that a startup (Copan Systems) had actually built a similar system a couple of years ago. They decided to take the conversation offline. Someone from Cornell pointed out that most disks die after switching on and off hundreds of times. In response, Sergey mentioned that they have optimized disks for this process but due to confidentiality, he cannot disclose the changes they have made.

A Self-Configurable Geo-Replicated Cloud Storage System

Masoud Saeida Ardekani, INRIA and Sorbonne Universités; Douglas B. Terry, Microsoft Research

Doug Terry presented Tuba, a geo-replicated key-value store that can reconfigure the sets of replicas when facing changes like access rate, user locations, etc., so as to provide better overall services. It's an extension work from Azure with several consistency model choices.

Doug started his presentation with a funny point about the recent shutdown of one of Microsoft Research's labs (in Silicon Valley, where he had been working). He posed the question, "What if someone decides to get rid of the data stored in California without any warning?" which would result in wrong configurations on all other clusters outside California. He proposed a solution based on a configuration service for such situations.

The aim of this service is to choose a better configuration for better overall utility, and to install a new configuration while clients continue reading and writing data. He presented Tuba, which extends Microsoft's Azure Storage and provides a wide range of consistency levels and supports consistency-based SLAs. Tuba's design is based on Pileus and maintains two sets of replicas: primaries and secondaries. Primaries are mutually consistent and completely updated at all times, while secondaries are lazily updated from primary replicas.

Doug then talked about how configuration selection and installation work. For instance, to select a configuration, it takes as input SLAs, read/write rate, latencies, constraints, cost, and the results of a configuration generator module. Applications can declare their acceptable level of consistency or latency and Tuba will generate all possible configurations satisfying the requested service, which is reasonable as the number of datacenters is usually small. Based on constraints defined by the application, such as cost model or data placement policies, the configuration manager will try to pick a configuration and put primary replicas for maximized consistency. Next, it will start moving data based on the new configuration. In this system, clients run in either Slow or Fast mode. Running in the Fast mode, clients read from the best replica and write data to all primaries. Running in the Slow

mode, clients do speculation for reading and then check the configuration to make sure data is read from a primary. In order to write data in Slow mode, clients should acquire a lock that guarantees no reconfiguration is in progress. He showed an example to demonstrate how to move the primary datacenter with Tuba.

Doug provided a quick evaluation setup. One cluster in the U.S., one in Europe, and one in Asia, and they used the YCSB benchmark to evaluate their system. He showed that the results on latency and utility were promising, and he also showed that Tuba can increase the overall number of strongly consistent reads by 63%.

A questioner asked about the way that Tuba takes into account future changes in client workloads. Doug mentioned this as the reason that reconfiguration improvements fade after some time. Another reconfiguration can solve the issue and its cost can be amortized.

f4: Facebook's Warm BLOB Storage System

Subramanian Muralidhar, Facebook; Wyatt Lloyd, University of Southern California and Facebook; Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, and Viswanath Sivakumar, Facebook; Linpeng Tang, Princeton University and Facebook; Sanjeev Kumar, Facebook

Sabyasachi Roy presented f4, an efficient warm BLOB storage system. Based on access patterns, warm BLOB content is isolated from hot content and f4 is used to store these contents. By being efficient, f4 lowers effective-replication-factor significantly and also provides fault tolerance in disk, host, rack, and datacenter levels. It's been deployed at Facebook and hosts a large amount of warm BLOB data.

Sabyasachi started his presentation with the definition of BLOB content, mentioning that most of the data stored in Facebook are photos and videos, which are immutable and unstructured. Moreover, these types of data cool down over time, making existing systems like Haystack less efficient for storing them. Basically, they split data into two rough categories, hot and warm. They do replication in various tiers to handle failures of disks, hosts, and racks. In order to make data access and recovery fast, their previous system stores 3.6 bytes for each byte of data. As this level of replication is too much for warm data, they tried to build a system that reduces space without compromising reliability. Using Reed-Solomon error correction coding and a decoder node to retrieve data and handle rack failures, they reduced the replication cost to 2.8x. Additionally, applying XOR on two BLOBs and storing the result in a third datacenter allows them to reduce the overhead down to 2.1x. In production, they consider hot data to be warm after three months or below the querying threshold of 80 reads/sec, then move it to a designated cluster designed for storing warm data.

Doug Terry highlighted the amount of data that would be transferred between datacenters when a failure happens. Sabyasachi mentioned that it would be a very rare event, but it might happen and they have already considered the bandwidth required for such situations. Someone from NetApp mentioned that one

of the charts in the paper does not make sense as 15 disks can saturate 10 Gb connections. They preferred to discuss this question offline. Finally, Jonas Wagner (EPFL) asked how f4 handles deletes. Sabyasachi replied that writes are handled by their old Haystack system, but a different system will take care of deletes.

Pest Control

Summarized by Jonas Wagner (jonas.wagner@epfl.ch)

SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems

Tanakorn Leesatapornwongsa and Mingzhe Hao, University of Chicago; Pallavi Joshi, NEC Labs America; Jeffrey F. Lukman, Surya University; Haryadi S. Gunawi, University of Chicago

Tanakorn Leesatapornwongsa pointed out that serious bugs hide deep within today's distributed systems and are triggered only by combinations of multiple messages and a specific ordering of events. Model checking has been proposed as a systematic solution for bug finding, but it cannot find the critical event interleavings in an exponentially large search space. Tanakorn presented SAMC, a tool that exponentially reduces the size of the search space through semantic knowledge about which event orderings matter for the application.

SAMC's users need to specify a set of rules (~35 lines of code for the protocols in SAMC's evaluation) that describe conditions where event order matters. SAMC evaluates these rules to prune unnecessary interleavings. Experiments show that this leads to speedups of 2–400x compared to state-of-the-art model checkers with partial order reduction. SAMC reproduces known bugs in Cassandra, Hadoop, and ZooKeeper, and also found two previously unknown bugs.

Ivan Beschastnikh (U. of British Columbia) asked whether relying on semantic information could cause bugs to be missed. Tanakorn replied that SAMC could not find those bugs that depended on non-deterministic behavior. SAMC also requires the developer-provided policies to be correct. Jonas Wagner wondered why SAMC did not find more previously unknown bugs. Tanakorn answered that this was because each bug requires a specific initial environment to be triggered, and this needs to be set up before running SAMC.

SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration

Pedro Fonseca, Max Planck Institute for Software Systems (MPI-SWS); Rodrigo Rodrigues, CITI/NOVA University of Lisbon; Bjørn B. Brandenburg, Max Planck Institute for Software Systems (MPI-SWS)

Concurrency bugs are hard to find and reproduce, because they need specific event interleavings to be triggered. Existing tools can explore possible interleavings for user-mode programs, but not in the kernel. Pedro Fonseca presented SKI, the first systematic approach for finding concurrency bugs in unmodified OS kernels. It runs the kernel and guest applications in a modified VMM, where each thread is pinned to a virtual CPU. By throttling these CPUs, SKI can exercise a diverse range of schedules.

SKI detects which CPUs/threads are schedulable by analyzing their instruction stream and memory access patterns. It uses

the PCT algorithm (ASPLOS 2010) to assign priorities to CPUs and systematically explore schedules. A number of heuristics and optimizations speed this up and are described in the paper. SKI supports several existing bug detectors to find data races, crashes, assertion violations, or semantic errors such as disk corruption.

SKI's authors used it to successfully reproduce four known bugs on several different kernels. This only takes seconds, because SKI explores 169k–500k schedules per second. SKI also found 11 new concurrency bugs in Linux file system implementations.

Stefan Bucur (EPFL) asked whether SKI could detect deadlocks. Pedro replied that they did not try this, although it is supported in a number of OSes that run on top of SKI. Bucur also asked how the effort of building SKI compares to the effort needed to instrument the kernel scheduler. Pedro pointed to related work, DataCollider, that, like SKI, avoided modifying the kernel because this was presumed to be very complicated. Srivatsa Bhat (MIT) asked about the maximum number of threads, to which Pedro replied that the number of virtual CPUs in QEMU (and thus SKI) is not limited.

All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Many applications, like databases and version control systems, employ techniques such as journaling, copy-on-write, or soft updates to keep their data consistent in the presence of system crashes. Yet their implementation is often incorrect. Thanumalayan Sankaranarayana Pillai identified file systems as the main cause for this, because they provide very weak guarantees that differ subtly for various configuration settings. He presented a study on file system guarantees and how these are used by applications.

Thanumalayan's tool, BOB (Block-Order Breaker), stress-tests a file system to produce an Abstract Persistence Model (APM), a compact representation of all the guarantees that have been observed not to hold. The companion tool ALICE (Application-Level Intelligent Crash Explorer) runs a user-provided workload, collects a system call trace, and uses the APM to enumerate possible intermediate states at various instants in this trace. If one of these states leads to inconsistent data after crash recovery, a bug has been found.

Alice found 60 such bugs, many of which lead to data loss. About half of these vulnerabilities are possible even for common file system configurations like btrfs.

Stephane Belmon (Google) asked what it meant for a “rename” system call to not be atomic. Thanumalayan explained that one can end up in a state where the destination file is already deleted but the source file is still present. Geoff Kuenning (Harvey Mudd College) asked for ideas for a better API that would make consistency easier for developers. Thanumalayan said related work had

attempted this, but no API other than POSIX is actually being used. POSIX is hard to use because there is no good description of the possible states after an API call. Emery Berger (Amherst) compared the situation to weak memory models. He asked if this is an API issue or a file system issue. Thanumalayan believes it's a combination of both. The current API is comparable to assembly language; a higher-level API would help. David Holland (Harvard) asked whether databases were more robust than other types of applications. Thanumalayan affirmed and said that databases in general fared better than version control systems.

Torturing Databases for Fun and Profit

Mai Zheng, Ohio State University; Joseph Tucek, HP Labs; Dachuan Huang and Feng Qin, Ohio State University; Mark Lillibridge, Elizabeth S. Yang, and Bill W Zhao, HP Labs; Shashank Singh, Ohio State University

Mai Zheng presented a system to test the ACID properties that we expect our database systems to provide. He showed that all examined database systems fail to guarantee these properties under some cases. At the heart of his work is a high-fidelity testing infrastructure that enforces a simple fault model: clean termination of the I/O stream at a block boundary. Because this model is simple (it does not consider packet reordering or corruptions), the errors it exposes are very relevant.

The system generates a workload that stresses a specific ACID property. It records the resulting disk I/O at the iSCSI interface level. It then truncates the iSCSI command stream at various instants to simulate an outage, performs database recovery, and scans the result for violations of an ACID property. The system augments the iSCSI command trace with timestamps, system calls, file names etc., and it uses this information to select fault points that are likely to lead to ACID violations. Such points are tried first to increase the rate at which problems are found. Once a bug is found, a delta debugging approach minimizes the command trace to narrow down the root cause of the problem.

The system was applied to four commercial and four open-source databases running on three file systems and four operating systems. It found ACID violations in all of them, especially durability violations. The speedups from using pattern-based fault point selection were instrumental for finding some of these bugs.

The first question concerned the configuration of the tested databases. Mai Zheng said that, whenever they were aware of options, his team configured the databases for maximum correctness. When asked why the work found mostly durability violations and few isolation violations, Mai Zheng explained that, although their workloads were designed to catch all types of violations, it is possible that isolation violations went undetected. Philip Bernstein (Microsoft Research) asked how likely the found bugs were to happen in practice. Mai Zheng replied that, in their traces, about 10–20% of the fault points led to a bug.

Award Announcements

Summarized by Rik Farrow (rik@usenix.org)

Rather than attempt to announce more awards while people were conversing during an outdoor luncheon, annual awards, as opposed to the ones specific to this particular OSDI, were announced before the afternoon break. Mona Attariyan (University of Michigan) received the SIGOPS Dennis Ritchie Doctoral Dissertation Award for her work on improving the troubleshooting and management of complex software. An ACM DMC Doctoral Dissertation Award went to Austin Clements of MIT for his work on improving database performance on multicore systems.

Steven Hand said that there would be no SIGOPS Hall of Fame awards this year. Instead, a committee will elect a large number of papers into the Hall of Fame at SOSP in 2015. Franz Kaashoek and Hank Levy will be committee chairs. At OSDI '16, they will focus on papers from 10–11 years previous to make things simpler. SOSP 2015 will be in Monterey and include an extra day for history. The first SOSP was in 1965.

Eddie Kohler won the Weiser Award for his prolific, impactful work on Asbestos, routing, and performing improvements in multicore databases among other things. Mark Weiser had asked that people make their code available, and Eddie has certainly done this, said Stefan Savage, who presented the award. Eddie also maintains HotCRP. Kohler, who wasn't present, had prepared a short video in which he said that what might not be obvious to people who know how cranky he is is that this community means a lot to him. Kohler thanked various people and the places where he had worked, like MIT, ICSI (Sally Ford), UCSD, Meraki (please use their systems). Kohler, now at Harvard, thanked everybody there, including Franz Kazakh, whom he described as close to an overlord, adding that it's lucky that he is so benevolent. Kohler ended by saying there are "a lot of white men on the Weiser award list. I am perhaps the only one that is gay. I hope we get a more diverse group of winners for the Weiser award."

Transaction Action

Summarized by Mainak Ghosh (mghosh4@illinois.edu)

Fast Databases with Fast Durability and Recovery through Multicore Parallelism

Wenting Zheng and Stephen Tu, Massachusetts Institute of Technology; Eddie Kohler, Harvard University; Barbara Liskov, Massachusetts Institute of Technology

Wenting motivated the problem by pointing to the popularity of in-memory databases due to their low latency. Unfortunately, they are not durable. Thus the goal of the work was to make an in-memory database durable with little impact on runtime throughput and latency. In addition, failure recovery should be fast. Wenting identified interference from ongoing transactions and slow serial disk-based recovery techniques as a major challenge. She proposed SiloR, which builds on top of Silo (a high performance in-memory database) and described how it achieves durability using logging, checkpointing, and fast disk recovery.

To make logging and checkpointing fast, SiloR uses multiple disks and multiple threads to parallel write. Wenting pointed out that SiloR logs values as opposed to operations because it enables parallel writes. SiloR also facilitates fast recovery because the need to preserve order among different log versions in operation logging is obviated. Recovery can be done in parallel as well. In the evaluation, Wenting showed the achieved throughput for SiloR to be less than vanilla Silo since some cores are dedicated for persistence. Checkpointing adds minimal overhead to the system. Recovery is also fast because SiloR can consume gigabytes of log and checkpoint data to recover a database in a few minutes.

Mark Lillibridge (HP Labs) asked about SiloR's performance if the system replicates logs and checkpoints. Wenting replied by admitting that replication is something that they are hoping to address in the future. Brad Morrey (HP Labs) asked about the lower per-core performance of SiloR in comparison to Silo. Wenting pointed out that SiloR does some additional work during logging and checkpointing which creates that difference. Brad's second question was about bottleneck during recovery. Wenting replied that SiloR tries to avoid being I/O bound by having multiple disk-based architecture.

Salt: Combining ACID and BASE in a Distributed Database

Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan, University of Texas at Austin

Chao started his talk by pointing out how transaction abstraction eases programming and is easy to reason about in ACID databases. Unfortunately, they are slow because providing isolation requires concurrency-control techniques like locks. In a distributed setting, complex protocols like 2PC make it even worse. The alternative, BASE, which does away with transactions for weaker forms of consistency, provides performance at the cost of code complexity. To bridge this gap, Chao proposed a new abstraction, Salt, which provides the best of both worlds. At this point, Chao made a key observation: following the Pareto principle, in a modern day application only a small set of transactions lead to performance limitation. This is because many transactions are not run frequently and a lot of them are lightweight. Their solution tries to BASE-ify this small set after identifying them.

Chao discussed the tradeoff between performance and complexity by using a bank transfer balance as an example. Splitting the transfer balance transaction such that deduction is in one half and addition is in another will lead to an inconsistent state being exposed for any transaction that tries to read the balance after the first transaction completes but before the second one begins. Since these transactions are critical, parallelizing them will lead to a lot of gains. Chao proposed BASE transactions, which consist of smaller alkaline transactions. Alkaline transactions can interleave with other alkaline transactions, but an ACID transaction cannot. This guarantees that the critical transaction provides the performance improvement without exposing inconsistent state to other transactions. The multiple granularities are provided by Salt isolation. Chao introduced three types

of locks: ACID, alkaline, and saline, which together provide the Salt isolation.

For evaluation, the whole abstraction was implemented on top of a MySQL cluster, and three workloads were used. Chao reported a 6.5x improvement in transaction throughput with just a single transaction BASE-ified. Thus, their goal for achieving performance with minimal effort while ensuring developer ease was met.

Marcos (Microsoft Research) asked about guidelines for developers on which transaction to BASE-ify. To identify a long-running, high-contention transaction, Chao proposed running the transaction with larger workloads and spot those whose latency increases. Marcos followed up by asking how to ensure BASE-ifying a transaction will not affect invariants like replication. To that Chao put the responsibility on the developer for ensuring this consistency. Henry (Stanford University) sought a clarification on the "Performance Gain" slide datapoints. Chao said it represented number of clients. Dave Andersen (CMU) asked about the future of their solution. Chao said he was very optimistic.

Play It Again, Sam

Summarized by Lucian Carata (lucian.carata@cl.cam.ac.uk)

Eidetic Systems

David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen, University of Michigan

David Devecsery introduced the concept of eidetic systems: systems that can "remember" all the computations they have performed (recording things like past memory states, network inputs and communications between processes), together with the relationships between the different pieces of data involved in those computations (provenance, or how data came to be—i.e., what inputs or data sources were involved in creating a given output).

David presented Arnold, an implementation of such a system based on deterministic record and replay techniques. During normal process execution, Arnold records sufficient information to allow for later replay at the granularity of a replay group (where a group is the set of threads/processes that share memory). At the same time, it maintains a dependency graph of inter-group data flows in order to support the tracking of provenance across replay groups. The more expensive operation of tracking the provenance of a piece of data within a group is left for the replay stage, when processes are instrumented using PIN to perform fine-grained taint tracking.

To reduce the storage overhead of recorded data, Arnold employs multiple data reduction and compression techniques (model-based compression, semi-deterministic time recording, and gzip). With those in place, the storage overhead for the typical utilization scenario (desktop or workstation machines) is predicted to be below 4 TB for four years.

Two motivating use cases were detailed, the first referring to tracking what data might have been leaked on a system with the

Heartbleed vulnerability, and the second covering the backward tracing for the source of an incorrect bibliographical citation (through the PDF viewer, LaTeX/BiBTeX processes and eventually to the browser window from where the data was copied). Forward tracing from the point of the mistake (in what other places was this wrong citation used?) is also possible using the same underlying mechanisms.

Fred Douglass (EMC) asked whether the system does anything to avoid duplication of data, such as copy-on-write techniques. David answered affirmatively. Arnold employs a copy-on-read-after-write optimization. As a follow-up, Fred asked whether the replay continues to work if the original inputs to a process are deleted. David replied that Arnold will continue to store those inputs for replay, employing deduplication and caches to reduce overheads. Ethan Miller (UC Santa Cruz) asked what happens when Arnold encounters programs that exhibit inherent randomness. David answered that they haven't found lots of programs with this behavior, but that such randomness would be treated as any other non-deterministic input that needs to be recorded in order to assure correct replay. Someone asked what happens if the users don't want provenance tracking for some pieces of data. David noted that Arnold itself can be used to determine all the places where a piece of data was used—and subsequently use that information to remove any trace of that data. Gilles Muller (INRIA) asked whether temporary files are kept, whether they are useful or a problem when trying to understand the source of information. David answered that temporary files are something like intermediate states (written out and read later). So Arnold will cache them in the file cache or regenerate them if needed.

Detecting Covert Timing Channels with Time-Deterministic Replay

Ang Chen, University of Pennsylvania; W. Brad Moore, Georgetown University; Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan, University of Pennsylvania; Micah Sherr and Wenchao Zhou, Georgetown University

Ang Chen presented the general idea behind covert timing channels, with an example of a compromised application sending normal packets over the network but encoding some sensitive data in the timing of those packets. The motivation behind the work in the paper is that existing state-of-art systems for detecting such covert timing channels look for specific statistical deviations in timing. However, those can be circumvented by attackers creating new encoding schemes or by altering the timing of just one packet in the whole encoding so that no statistical deviation exists.

The proposed solution relies on determining the expected timing of events and then detecting deviations from it. The insight is that instead of predicting the expected timing (a hard problem), one can just reproduce it using record and replay techniques: recording the inputs to an application on one machine and replaying them on a different one.

However, Ang explained that existing deterministic replay systems are not sufficient for the stated purpose, as they reproduce functional behavior of an application, but not its timing behavior (e.g., XenTT shows large time differences between actual program execution and replay). In this context, time-deterministic replay is needed. To achieve this, various sources of “time noise” must be handled.

During the presentation, the focus was placed on time noise generated by different memory allocations and cache behavior. The solution presented for that problem aims to maintain the same access patterns across record and replay. This is achieved by flushing caches before record and replay and by managing all memory allocations to place all variables in the same locations in both cases.

A prototype of time-deterministic replay, Sanity, was implemented as a combination of a Java VM with limited features and a Linux kernel module. The evaluation of this prototype shows that Sanity managed to achieve a very small timing variance across multiple runs of a computation-intensive benchmark (max 1.2%) and provided accurate timing reproduction (largest timing deviation of 1.9%) on a separate workload. Sanity was also able to detect timing channels without any false positives or false negatives.

John Howell (Microsoft Research) asked about the scope of attacks being considered, and in particular, whether an attack payload present in the input stream wouldn't still be replayed by Sanity on the reference machine. Ang answered that for the replay, the system has to assume that the implementation of the system is correct. One of the creators of XenTT noted that XenTT provided a fairly accurate time model (in the microsecond range): Was that model used for the XenTT results? Ang answered that the same type of replay was used, and the results were the ones shown during the presentation. Gernot Heiser (University of New South Wales and NICTA) questioned the feasibility of the given solution in realistic scenarios since it relies on the existence of a reference (trusted) machine that is identical to the machine running the normal application. In this case, the application could be run directly on the trusted machine. Ang acknowledged that the current solution requires a correct software implementation and an identical machine but pointed out that there are still multiple situations where the solution might be feasible and where such identical machines exist (e.g., a datacenter).

Identifying Information Disclosure in Web Applications with Retroactive Auditing

Haogang Chen, Taesoo Kim, Xi Wang, Nikolai Zeldovich, and M. Frans Kaashoek, MIT CSAIL

Haogang Chen started his presentation by highlighting the recurrent cases of data breaches leading to user data being compromised as a result of Web site vulnerabilities. Multiple solutions deal with preventing such breaches, but not with damage control solutions after a breach occurs.

However, Haogang observed that even if vulnerabilities exist, they might not be exploited, or the attackers might not actually steal all the data they obtain access to. Therefore, an important goal would be to precisely identify breached data items.

The state-of-the-art in this respect requires logging all accesses to sensitive data, and careful inspection after an intrusion, but this is often impractical. An alternative is presented in Rail, a system that aims to identify previously breached data after a vulnerability is fixed. The insight used is that Web application requests/responses can be recorded during normal execution and then replayed after an administrator fixes a vulnerability. The difference in data being sent can be attributed to the information that was leaked due to that vulnerability.

Compared to other record/replay systems, the challenge in implementing Rail is minimizing the state divergence on replay as that might lead to the reporting of false positives. The proposed solution assumes that the software stack below the application is trusted, and consists of an API for Web application developers facilitating deterministic record/replay and data identification at the object level.

The design revolves around the notion of action history graphs: An action is generated for each external application event (e.g., user request, timer), and all application code triggered by that event is executed in the context of the action. Any objects used in that code are connected to the action, resulting in a history graph. This is then used to replay each action in time order, whenever one of its inputs/outputs has changed.

Haogang also discussed the case of replay in the presence of application code changes and non-deterministic inputs. The chosen example involved changes in the components of an array (list of admins), which in turn invalidated replay data associated with particular indexes in the array (e.g., password assignments). Rail provides an input context object that can be used for associating such data with particular stable object keys.

In the evaluation of Rails, Haogang highlighted that it performed better than log-inspection approaches, giving no false negatives and (for the tested workloads) only one false positive, the result of a malicious account created by the attacker. In terms of replay, Rails needed to do so only for the fraction of requests related to the attack (max 10.7%). Overall throughput overhead varied between 5% and 22%.

Stefan (Google) raised the issue of the threat model being considered; in particular, the fact that the application must trust the Web framework and other software stack components. If one manages to inject random code into the process, logging might be bypassed. Haogang acknowledged that that is the case, as the assumption is that the framework is trusted.

Help Me Learn

Summarized by Xu Zhao (muk.zhao@mail.utoronto.ca) and Sankaranarayanan Pillai (madthanu@gmail.com)

Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, Karthik Kalyanaraman, Microsoft Research

Trishul began by introducing machine learning and deep learning. Deep learning differs from other kinds of machine learning by not requiring a human to extract the features of the training data. Deep learning can automatically learn complex representations (without requiring humans); an example utility is computer vision. Deep learning algorithms can be thought of as a network of multiple levels of neurons that work level-by-level. For example, in computer vision, initial levels identify simple representations such as color and edges, while higher levels automatically learn complex representations such as edges and textures. The accuracy of deep learning can be improved by increasing the size of the model (such as the number of levels in the deep learning network) and by increasing the amount of data used for training the model. Both of these require better, scalable systems.

The authors proposed Adam, a scalable deep learning system. Adam contains three parts: a data server (that supplies data to the models), a model training system (where multiple models learn), and a model parameter server (where all models store their learned weights). The overall design aims at data parallelism and model parallelism: a large data set is divided, and each part is simultaneously used for training, with multiple models also trained at the same time. Adam uses an asynchronous weight update technique, where learned weights are propagated to the model-parameter server slowly (the weight update operation is both commutative and associative). To make the models distributed, Adam partitions the models to fit a single machine; the working version of the model is fit into the L3 cache, so that memory is not a bottleneck. Furthermore, Adam optimizes communication with the model-parameter server by asynchronous batching.

The authors evaluated the accuracy of Adam using MNIST as a baseline. By turning on asynchronization, Adam gets tremendous improvement; asynchronization can help the system jump out of local minimum. On a computer vision task, Adam has twice the accuracy of the world's best photograph classifier, because it can use bigger models and a larger training data set.

John Ousterhout (Stanford) asked about the size of the model in terms of bytes and how much data is shared among how many models. Trishul answered that, usually, 20 to 30 models share terabytes of data. A student from Rice University asked how hardware improvements can help (can bigger models be combated with bigger hardware?). Trishul answered that bigger models are harder to train; hence, hardware improvement does not simply solve the problem. Another question concerned the number of machines assigned for the model replica and how a replica fits into the L3 cache. Trishul answered that there are

four machines assigned to model replicas and only the working set of the model needs to fit into the L3 cache.

Scaling Distributed Machine Learning with the Parameter Server

Mu Li, Carnegie Mellon University and Baidu; David G. Andersen and Jun Woo Park, Carnegie Mellon University; Alexander J. Smola, Carnegie Mellon University and Google, Inc.; Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su, Google, Inc.

Mu began by showing users a real-world example of machine learning: ad-click analysis, similar to Google's advertisements on its search results. While machine-learning accuracy can be improved by increasing the size of the model and by increasing the amount of data, dealing with huge model sizes and data requires distributing the work among multiple machines.

In the presented system, the training data is fit into worker machines, while the model is fit into server machines. Worker machines compute gradients and push them to the servers, which compute and update the model; the model is then pulled by the worker machines. The authors found that accessing the shared model is costly because of barriers and the network communication. To reduce the cost, the authors introduce the concept of a Task. Each Task contains separate CPU-intensive and network-intensive stages, and performance can be increased by running the stages asynchronously. The system also allows users to trade off consistency for reduced network traffic; waiting time is eliminated by relaxing consistency requirements. Mu also briefly talked about other features like user-defined filters, explained further in the paper. Apart from performance, the system achieves practical fault tolerance using consistent hashing on model partitions. By only replicating the aggregating gradient, the system reduces network traffic (while, however, introducing CPU overhead). Also, the system exposes the output using a key-value API.

For the evaluation, the authors ran sparse logistic regression with 636 terabytes of real data on 1,000 machines with 16,000 cores in total. The system outperformed two baselines; furthermore, the waiting time of training can be eliminated by relaxing the consistency requirement. Mu also presented the result of running another application, Topic Model LDA: increasing the number of machines from 1,000 to 6,000 provided a 4x speedup. Finally, Mu showed the results with 104 cores; the key here is the tradeoff between network communication and consistency.

A student from NYU asked about more quantitative details of the tradeoff between consistency and accuracy. Mu answered that it really depends on the model and the training algorithm, and that he authored an NIPS paper showing the theoretical upper-bound. Kimberly Keeton (HP Labs) asked why the authors chose Boundary Delay instead of other consistency models; Mu answered that Boundary Delay was just one of the consistency models they used; they actually used different consistency models for different applications. A questioner from Microsoft Research asked what accuracy meant in the y-axis of the accuracy graph; Mu answered that when the model gets the accuracy quantity needed, they will stop training.

GraphX: Graph Processing in a Distributed Dataflow Framework

Joseph E. Gonzalez, University of California, Berkeley; Reynold S. Xin, University of California, Berkeley and Databricks; Ankur Dave, Daniel Crankshaw, and Michael J. Franklin, University of California, Berkeley; Ion Stoica, University of California, Berkeley and Databricks

Joseph began the talk by explaining how, in modern machine learning, combining two representations of the data, tables and graphs, is difficult. For tables, there are already many existing solutions, like Hadoop and Spark. For graphs, we have GraphLab and Apache Graph. However, users have to learn both table and graph solutions, and migrating data between them is difficult. GraphX unifies tables and graphs; the authors show that there is a performance gap between Hadoop/Spark and GraphX, indicating GraphX is really needed.

Joseph showed how GraphX converts graphs into table representation, and how it represents graph operations (like gather and scatter) into table operations (like Triplet and mrTriplet). The authors did many optimizations on the system, such as remote caching, local aggregation, join elimination, and active set tracking.

The authors evaluated GraphX by calculating connected components on the Twitter-following graph; with active vertex tracking, GraphX got better performance, while with join elimination, GraphX decreased data transmission in the combining stage. The evaluation was done by comparing the performance between GraphX, GraphLab, Giraph, and naive Spark. GraphX is comparable to state-of-the-art graph-processing systems.

Greg Hill (Stanford University) asked how a graph can be updated in GraphX; Joseph answered that, currently, GraphX doesn't support updating. Schwarzkopf (Cambridge) asked why there are no evaluations and comparisons between GraphX and Naiad; Joseph answered that the authors found it difficult to express application details in Naiad. The third questioner asked whether ideas in GraphX can be backported into existing systems; Joseph answered that some techniques can be backported, but many techniques are tied to GraphX's particular design.

Hammers and Saws

Summarized by Ioan Stefanovici (ioan@cs.toronto.edu)

Nail: A Practical Tool for Parsing and Generating Data Formats

Julian Bangert and Nickolai Zeldovich, MIT CSAIL

Julian motivated Nail by describing the problems of binary data parsers today: Most of them are handwritten and highly error-prone (indeed, a number of recent parsing bugs generated high-profile security vulnerabilities in SSL certification, and code signing on iOS and Android applications). One option is to employ parser generators like Bison to generate a parser, but this would still involve extra handwritten code by the programmer to manipulate the parsed data in the application, and output it back to the binary format. In addition, this approach cannot handle non-linear data formats (such as ZIP archives). With Nail, programmers write a single grammar that specifies both the format of the data and the C data type to represent it, and Nail will cre-

ate a parser, associated C structure definitions, and a generator (to turn the parsed data back into a string of bytes).

The Nail grammar supports standard data properties (size, value constraints, etc.), but it also reduces redundancy introduced by having multiple copies of the same data by including the notion of dependent fields (values that depend on other values). Non-linear parsing (e.g., parsing a ZIP archive backwards from the header field) is supported using “streams”: multiple paths that can each be parsed linearly. For unforeseen stream encodings (e.g., parsing dependent on arbitrary offset and size fields), Nail provides a plugin interface for arbitrary programmer code (which would be much smaller than an entire parser). Output generation back to binary format is not a pure bijection but, rather, preserves only the semantics of the specification, allowing Nail to discard things like padding and other redundant data.

Nail is implemented for C data types. The code generator is implemented using Nail itself (100 lines of Nail + 1800 lines of C++). To evaluate Nail’s ability to handle real formats, Julian implemented various grammars: Ethernet stack (supporting UDP, ARP, ICMP), DNS packets, and ZIP archives. In all cases, the implementation with Nail consisted of many fewer lines of code than a handwritten alternative, and captured all the complexities of each respective data format. A Nail-generated DNS server also outperformed the Bind 9 DNS server in queries/sec. Nail builds on previous work in the area (e.g., the basic parsing algorithm is the Packrat algorithm described in Bryan Ford’s MSc thesis).

Eddie Kohler (Harvard University) remarked that memory usage is a known disadvantage of Packrat parsers, and asked whether that was a valid reason to continue using handwritten parsers instead. Julian replied that for most parsing that does not involve backtracking (such as DNS packet parsing), memory usage and performance is not a concern.

lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems

Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm, University of Toronto

Yongle opened by discussing a critical problem of distributed systems: performance anomalies. Such anomalies increase user latency and are very hard to understand and diagnose, since they typically involve understanding the flow of a request across many nodes in a distributed system. Existing solutions for distributed request profiling (MagPie, X-Trace, Dapper, etc.) typically involve intrusive, system-specific instrumentation. A key observation is that most distributed systems today already generate TBs of log data per day, much of which consists of information about the flow of requests through the system (since developers rely on this information for post-mortem, manual debugging). *lprof* is a non-intrusive profiler that infers request control flow by combining information generated from static analysis of source code with parsing of runtime-generated system logs. Yongle presented a sample “latency over time” graph generated by *lprof* that showed unusually high latency for a writeBlock request in HDFS. Combined with per-node latency

information (also generated by *lprof*), the problem can conclusively be attributed to unnecessary network communication.

lprof generates a model by performing static analysis on application byte code. This model is then used while performing log analysis at runtime (using a MapReduce job) to profile request flow and save the information into a database (e.g., for use later in visualization). Challenges involved in log analysis include: interleaved messages from different request types, lack of perfect request identifiers, and log entries generated by the same request spread across several machines. In order to trace the flow of a request, *lprof* needs to identify a top-level method (that starts to process the request) and a request identifier (that is not modified during the request) and maintain log temporal order. The model generated by static byte code analysis is used during the log analysis to solve all these problems. The key intuition is that unique request identifiers are already included by developers in logs for manual post-mortem debugging. Cross-node communication pairs are identified as socket or RPC serialize/deserialize methods. Temporal order is inferred by comparing the output generated for a specific request with the order of the corresponding source code that generated it.

lprof was evaluated on logs from HDFS, Yarn, HBase, and Cassandra using HiBench and YCSB as workloads. *lprof* grouped 90.4% of log messages correctly; 5.7% of messages involved request identifiers that were too complicated for *lprof* to handle, while 3% of logs could not be parsed, and 1% of log messages were incorrectly grouped. *lprof* was also helpful in identifying the root cause of 65% of 23 real-world performance anomalies (the 35% where it was not helpful was due to insufficient log messages).

Rodrigo Fonseca (Brown University) asked what developers could add to their logging messages to help *lprof* work better. Yongle replied that better request identifiers would help. Rodrigo further asked how this could be extended to combine information across multiple applications. This remains as future work. A developer of HDFS mentioned that they log per-node metrics (e.g., bytes processed/sec) at runtime, and was wondering how *lprof* can be used with this information for performance debugging. Yongle replied that *lprof* would provide the cross-node request tracing, while the per-node metrics could complementarily be used for finer-grained debugging. Someone from North Carolina State University wondered how block replication in HDFS would affect the analysis performed by *lprof*, but the discussion was taken offline.

Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud

Stefan C. Müller, ETH Zürich and University of Applied Sciences Northwestern Switzerland; Gustavo Alonso and Adam Amara, ETH Zürich; André Csillaghy, University of Applied Sciences Northwestern Switzerland

Stefan began by describing the motivation for their work: large-scale astronomy data processing. Astronomers enjoy coding in Python due to its simplicity and great support for numeric and graphing libraries. *Pydron* takes single-threaded code written by astronomers and semi-automatically parallelizes it, deploys it on EC2 or private clusters, and returns the output to the astronomer.

Astronomers can't just use MapReduce (or similar big-data "solutions"). Compared to the large-scale commercial systems for big-data processing with thousands of deployments and millions of users, astronomy projects typically have just a single deployment, and fewer than 10 people. Furthermore, that single deployment involves never-seen-before data processing and analysis on never-seen-before kinds and volumes of data. Code reusability across such projects is very limited. Astronomers are also developers, not users: They continuously iterate the logic in the analysis code and rerun experiments until they obtain publishable results and never run the code again. The goal of Pydron is not only to reduce total execution time, but also decrease the amount of time spent by astronomers writing code (i.e., writing sequential code involves much less effort than a highly parallel MPI program).

Most astronomy code is broken up into two types of functions: those that "orchestrate" the overall analysis and those called from the orchestrator function to compute on data, with no global side-effects. Pydron only requires "orchestrator" functions to be annotated with a "@schedule" decorator, and compute functions with a "@functional" decorator. At runtime, upon encountering a @schedule decorator, Pydron will start up EC2 instances and Python processes, transfer over all the code and libraries, schedule the execution, and return the results back to the user's workstation (as if the code had been executed locally).

A random forest machine learning training example shows almost-linear speedup in execution time with an increasing number of cores using Pydron. Pydron generates data-flow graphs for functions annotated with the "@schedule" decorator, and dynamically updates the data flow graph at runtime with information that is unknown statically in Python (data-dependent control flow, dynamic type information, invoked functions, etc.). The changing data flow graph then informs decisions about the degree of parallelization possible from the code. The runtime of exo-planet detection code parallelized with Pydron decreased from 8.5 hours on a single EC2 instance down to 20 minutes on 32 instances, significantly reducing the turnaround time for astronomer tasks. Future work includes better scheduling algorithms, data-specific optimizations, pre-fetching, and dynamic resource allocation.

Brad Morrey (HP Labs) complimented the effort of improving the workflow of non-computer scientists by doing good systems work, but wondered where Pydron's approach of graph-based decomposition fails (and where performance is poor). Stefan answered by admitting that some algorithms are inherently sequential (and parallelization is not possible), and the system is designed for coarse-grained parallelization (where compute-intensive tasks take seconds/minutes), so that's where it sees the most benefit. Another limitation is that the system is currently designed for functional algorithms (that don't require changes to global mutable state). Stefan (Google) wondered whether the images in the example (specified by paths) were stored as files somewhere, and whether large data sets would limit the use of

Pydron. Stefan admitted that Pydron currently uses Python's native serialization library (pickle) and sends objects over TCP, and there is room for future work. Scott Moore (Harvard University) asked whether the authors had looked into validating functional annotations dynamically (using graph updates). Stefan said that at the moment, no checks take place, but that it would be useful for developers to find bugs as well. Someone from Stanford clarified Pydron's assumption that each function must be able to run independently (no side-effects) in order for it to be parallelized and wondered how Pydron would work with stencil functions. Stefan replied that the inputs to the function would need to be passed as arguments (respecting the "no side-effects" rule).

User-Guided Device Driver Synthesis

Leonid Ryzhyk, University of Toronto, NICTA, and University of New South Wales; Adam Walker, NICTA and University of New South Wales; John Keys, Intel Corporation; Alexander Legg, NICTA and University of New South Wales; Arun Raghunath, Intel Corporation; Michael Stumm, University of Toronto; Mona Vij, Intel Corporation

Leonid began by remarking that device drivers are hard to write correctly, hard to debug, and often delay product delivery (as well as being a common source of OS failures). A key observation motivating the work is that device driver development is a very mechanical task: It boils down to taking an OS interface spec and a device spec and proceeding to generate the driver without much freedom into how the driver can interact with the device. In principle, this task should be amenable to automation. Leonid approached the device driver synthesis problem as a two-player game: driver vs. (device + OS). The driver is the controller for the device, and the device is a finite state machine where some transitions are controllable (triggered by the driver), while others are not controllable (e.g., packet arriving on the network, error conditions). The problem can be seen as a game, where the driver plays by making controllable actions, and the device plays by making uncontrollable actions. A winning strategy for the driver guarantees that it will satisfy a given OS request regardless of the device's possible uncontrollable actions.

Leonid used a LED display clock as an example. A driver that wants to set a new time on the clock to an arbitrary time must first turn off the internal oscillator of the clock to guarantee an atomic change of all the hour, minute, and second values (without oscillator-induced clock ticks changing the time in between operations). All the possible strategies are considered, and by backtracking from the winning goal (in which the new time is set correctly on the clock) to the current, initial state, you can find all the winning states in the game. Leonid proceeded to demo his device driver synthesis tool (called Termite) and showed how the tool generated code to change the time on the clock according to the winning strategy.

Crucially, Termite is a "user-guided" synthesis tool (in contrast to "push-button" synthesis tools, that generate the entire implementation automatically). The rationale is to keep developers in charge of important implementation decisions (e.g., polling vs. interrupt). The driver synthesis tool then works as

a very smart auto-complete-like tool (smart enough to generate the entire driver for you!), but maintains correctness in the face of arbitrary user code changes (and will inform the user if their proposed code changes cannot lead to a winning strategy). Termite also serves as a driver verification tool for a manually written driver.

Leonid then addressed the problem of specification generation for OS and devices: If developing the specifications for synthesis takes longer than writing the driver by hand, is it worth it? Two important observations are that OS specs are generic (i.e., made for a class of devices) and can often be reused, and device specs from the hardware development cycle can be obtained from hardware developers and used for synthesis. Some drivers synthesized to evaluate Termite include drivers for: a real-time clock, a Webcam, and a hard disk. Leonid then addressed the limitations of the work: the focus is currently on synthesizing control code (none of the boilerplate resource allocation code), it is single-threaded (current work-in-progress focuses on multithreading the generated code), and it does not currently provide DMA support (also work-in-progress).

Ardalan Amiri Sani (Rice University) wanted to know how difficult it would be to figure out what the code that Termite generated is doing. Leonid explained that the kind of expertise you need to use Termite is the same you need to write a driver. There are tools that can help you make sense of what instructions are doing to the device, but at the end of the day you need to understand what you're doing. Someone from INRIA wondered whether it would be interesting for the tool to generate a Devil specification that described what the device does instead of generated code. Leonid replied that a better strategy would be to change the Termite specification language to include Devil-style syntax. Joe Ducek (HP Labs) wondered about the difficulty in getting DMA support and whether it would be possible to have the developer handle the DMA bits and let Termite do the rest. Leonid replied that Termite currently supports user-written DMA code in addition to the Termite-generated code. Automatically generating driver code for DMA is difficult because it generates a state explosion in the game-based framework. Brad Morrey (HP Labs) asked how exploring the state space to solve the two-player game scales. Leonid replied that a big part of the whole project was implementing a scalable game solver, and the results are published in a separate publication.

2014 Conference on Timely Results in Operating Systems

October 5, 2014, Broomfield, CO

Summarized by Timo Hönig, Alexander Merritt, Andy Saylor, Ennan Zhai, and Xu Zhang

Memory Management

Working Set Model for Multithreaded Programs

Kishore Kumar Pusukuri, Oracle Inc.

Summarized by Xu Zhang (xzhang@cs.uic.edu)

Kishore opened his talk with the definition of working set size (WSS) of multithreaded programs, which is the measure of the number of pages referenced by the program during a certain period of time multiplied by the page size. Knowing the WSS helps provide insight into application memory demand and is useful for dynamic resource allocation: for example, the page replacement algorithm in operating systems. Various approaches for approximating WSS exist, including simulation-based and program traces-based techniques. However, they only work on single-threaded programs, and more importantly, such measurements are too expensive to be applicable for effective resource management.

Characterizing WSS is also non-trivial. Not only does WSS vary from application to application, it is also affected by several factors. The author collected data from running 20 CPU-bound multithreaded programs on large scale multicore machines and identified four factors—what he denotes as “predictors”—that correlate to WSS: resident set size (RSS), the number of threads, TLB miss rate, and last-level cache (LLC) miss rate. To increase prediction accuracy and reduce the cost of approximation, Kishore further refined the predictors by pruning the ones of less importance using Akaike information criterion (AIC), avoiding overfitting and multicollinearity at the same time.

The resulting major predictors are RSS and TLB miss per instruction. Based on them, three statistical models were developed using supervised learning: linear regression (LR), K nearest neighbor (KNN), and regression tree (RT). These models are further selected using cross-validation tests, with KNN being the most accurate, which enjoys an approximation accuracy of 93% and 88% by normalized root mean squared error (NRMSE) on memcached and SPECjbb05. Notably, the developed model has very little overhead, which is in granularity of microseconds compared to hours.

Kishore also briefly talked about their ongoing work for WSS-aware thread scheduling on large scale multicore systems. He explained two existing scheduling algorithms, grouping and spreading, both using the number of threads as a scheduling metric. Grouping gangs threads together on a few cores initially and spreads them out if the number of threads exceeds the limit. By contrast, spreading distributes all threads uniformly across all cores and sockets at the start of the day. The author argued that using the number of threads for thread scheduling is not sufficient on the target system and illustrated this with two examples.

Ken Birman (Cornell) noted that approximating WSS is less important as memory has become larger and asked whether there were other contexts where such a machine-learning approach might be usable. Kishore replied yes and pointed out an application in virtual machine allocation, scheduling, finding failures, and providing high availability in the cloud. Ken followed up asking whether the author had the same findings in those cases where a small subset of the available metrics become dominant and are adequate for training. Kishore replied yes. The second questioner asked whether the errors are under- or overestimated. The author said it doesn't matter since WSS varies greatly from app to app. Someone asked what accuracy is acceptable for an application using such models. Based on his understanding of the literature, Kishore noted that above 80% is considered good. The final question was architecture related: Why not use a translation storage buffer (TSB) in the SPARC architecture, which caches the most recently used memory mappings, for WSS or data migration decisions? Kishore said their work is based on the proc file system without any kernel modifications.

MLB: A Memory-Aware Load Balancing for Mitigating Memory Contention

Dongyou Seo, Hyeonsang Eom, and Heon Y. Yeom, Seoul National University
Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Dongyou Seo began by illustrating how modern CPUs are manycore and that future chips are envisioned to continue this trend; he referred to a forward-looking statement made by Intel for 1,000-core chips. To maximize the use of all cores, many systems accept more work by becoming multi-tenant, as is the case in server systems. Such scenarios, however, expose applications to possible contention on resources shared by cores, such as the last-level cache, and the limited bus bandwidth to local DRAM. A challenge, then, is to efficiently manage these shared resources to limit the effects of contention on applications, e.g., prolonged execution times. Existing OS task schedulers, such as in Linux, manage multicore chips by migrating tasks between cores, using the measured CPU load that a task places on a core as an important metric for load-balancing, while not prioritizing, or fully ignoring, the impact of memory bandwidth contention on task performance. The authors argue that memory bandwidth is one of the most important shared resources to understand, since the ratio between the number of cores and available bandwidth per core is increasing across generations of processors.

To address this challenge, their work presents a memory contention-aware task load balancer, “MLB,” which performs task migration based on an understanding of a task’s memory-bandwidth pressure. Their work contributes a memory-bandwidth load model to characterize tasks, a task scheduling algorithm using this model to influence when/where tasks are migrated among all processors, and an implementation of the first two contributions in the Linux CFS scheduler. An analysis was extended to compare against Vector Balancing and Sorted Co-scheduling (VBSC) and systems hosting a mix of CPU- and GPU-based applications.

Their contention model is defined by the amount of retired memory traffic measured by memory request events from last-level cache misses and prefetcher requests. Because each application’s performance may be differently affected by the same level of contention, a sensitivity metric is defined for each application. Together, both metrics are used by an implementation of MLB in the Linux CFS scheduler. Tasks are grouped based on those that are highly memory intensive and those that are not. Two task run-queue lists are used (one for each category of task), with a single run queue assigned to a given core. Tasks are migrated between cores via the run queues when indicated by the memory contention model. An evaluation of their methods included a comparison with a port of the VBSC model into a modern version of Linux for a variety of applications from the SPEC benchmark suite as well as TPC-B and TPC-C. An extension of their work supports multithreaded applications. NUMA platforms, however, were not examined in this work.

Someone asked when their task migration mechanisms are triggered. Dongyou replied that a burn-in time is required to collect sufficient information for the model to stabilize (to distinguish memory intensive vs. non-memory intensive tasks). A second questioner addressed the extension of MLB to multithreaded tasks, asking whether the techniques in the paper would still apply. Dongyou responded that their methods can increase the hit rates of the last-level cache and that he plans to optimize them further. Following up, the questioner suggested it would be interesting to compare MLB to a manually optimized task-pinning arrangement to illustrate the gains provided by models presented in the paper.

A final question similarly addressed the lack of analysis on NUMA platforms, asking what modifications would be necessary. The response suggested that they would need to observe accesses to pages to understand page migration strategies, as just migrating the task would not move the memory bandwidth contention away from that socket.

Cosh: Clear OS Data Sharing in an Incoherent World

Andrew Baumann and Chris Hawblitzel, Microsoft Research; Kornilios Kourtis, ETH Zürich; Tim Harris, Oracle Labs; Timothy Roscoe, ETH Zürich
Summarized by Xu Zhang (xzhang@cs.uic.edu)

Kornilios started the talk by justifying the multikernel model—treating the multicore system as a distributed system of independent cores and using message passing for inter-process communication—a useful abstraction to support machines that have multiple heterogeneous processors. He illustrated with an example of their target platform—the Intel MIC prototype on which the Intel Xeon Phi processor is based. It has four NUMA domains or three cache-coherent islands, and transfers data with DMA between islands. Such heterogeneity breaks existing OS assumptions of core uniformity and global cache-coherent shared memory in hardware architecture. Multikernel models fit nicely since they treat the operating system as a distributed system.

One problem with the multikernel model, however, as Kornilios pointed out, is that there is no easy way to share bulk data, either for I/O or for large computation. Lacking support for shared memory, the multikernel model forces data copying to achieve message passing. This is the gap that coherence-oblivious sharing (Cosh) tries to close—to share large data with the aid of specific knowledge of underlying hardware. Cosh is based on three primitive transfers: move—exclusive read and write transfer from sender to receiver; share—read sharing among sender and receiver; and copy—share plus exclusive receiver write. And no read-write sharing is allowed by design. To make bulk sharing practical, two additional features—weak transfer and aggregate—are built on top of the primitives. Weak transfer allows the sender to retain write permission and to defer or even neglect permission changes. It is based on the observation that changing memory permission is costly and is not always necessary—for example, if the transfer is initiated from a trusted service. Aggregate provides byte granularity buffer access since page granularity doesn't work for everything, particularly handling byte data. Aggregate exports a byte API by maintaining an aggregate structure on top of page buffers. Kornilios illustrated the API with examples resembling UNIX pipes and file systems.

A prototype of Cosh was implemented on top of the Barrelfish operating system, which is an instance of the multikernel model. The prototype supports MIC cores natively. Kornilios showed weak transfer being a useful optimization for host-core transfers. And although pipelining helps, host-to-MIC transfers are bogged down with high latency. Kornilios also demonstrated results of an “end-to-end” evaluation of replaying panorama stitching from traces captured on Linux. While Cosh enjoys the same latency of Barrelfish's native file system on host-core transfers, the Cosh prototype still suffers from the high-latency of DMA transfers between MIC and host cores. With the help of perfect cache, the latency is reduced by a factor of 20 but is still 17 times greater than host-to-host copying latency. The major bottleneck is due to slow DMA transfers and slow co-processors.

Someone asked how the performance improved by using cache in host-to-MIC core transfers. Kornilios explained that the cache they implemented was a perfect cache and could be as large as possible, which reduced the number of DMA operations. The second questioner asked whether they are planning to support GPU. Kornilios said they haven't looked at GPU particularly because of the lack of support for access permissions. But GPUs are becoming more general purpose, so he is personally optimistic. Another question was whether they had explored the multiple writer concurrency model. Kornilios replied no, because there is no read-write sharing in Cosh by design. He further commented that write sharing is difficult for programmers to reason about, and cache-coherent and shared memory is hard to think about. He brainstormed that it might be feasible if data partitioning was provided or if versioning was available. The last questioner asked whether the Cosh model is used for synchronization. Kornilios answered no, since the multikernel is based on message passing and there is no shared memory.

System Structuring

Summarized by Andy Saylor (andy.saylor@colorado.edu)

Fractured Processes: Adaptive, Fine-Grained Process Abstractions

Thanumalayan Sankaranarayanan Pillai, Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, University of Wisconsin—Madison

Pillai opened by discussing the traditional OS process abstraction and noting that most applications are monolithic (single process). This poses problems when managing or debugging applications: for example, we are unable to restart just part of a process if it crashes or is upgraded but must restart the whole thing (potentially affecting GUIs and other components). Likewise, many existing debugging tools like Valgrind produce too much overhead when run on a large process, making such tools unfeasible to use in production (e.g., to monitor sensitive parts of an application for memory leaks). To counter these issues, the authors propose Fracture, a mechanism for subdividing applications into an isolated set of processes communicating via RPC. Fracture is designed to work with C programs and allows the user to flexibly isolate only the parts of the program that require it, minimizing the overhead incurred by splitting a monolithic program into multiple processes.

Using Fracture, a developer divides their program into logically organized modules (collections of functions) using annotations in the code. The developer then defines a mapping of each module into an associated FMP (fractured mini-process) that will run as an isolated process. Each FMP can accommodate one or more modules. Developers can dynamically reconfigure their applications to run in a single FMP (i.e., as they would have run without Fracture), or they can split the application into multiple FMPs to troubleshoot specific bugs, isolate sensitive code, etc. Fracture is also capable of using a min-cut algorithm and a list of developer-defined rules regarding which module must be isolated from which other modules to automatically partition a program into an optimal set of FMPs. Each FMP acts as a micro-server, responding to RPC requests from other FMPs. In order to facilitate this, the developer must adhere to certain rules: no sharing of global state between modules, annotation of pointers so the data they point to may be appropriately serialized, etc.

The authors tested Fracture against a handful of existing applications: Null-httpd, NTFS-3G, SSHFS, and Pidgin, the universal chat client. The overhead imposed by using Fracture depends on the specific mapping of modules to FMPs chosen by the developer. In the base case where all modules map to a single FMP, Fracture obtains effectively native performance. At the other extreme, where each module is mapped to its own FMP, performance degrades significantly, from 10% of native performance to 80% of native performance depending on the application. Using Fracture's intelligent partitioning to automatically produce an optimal map of modules to FMPs minimizes performance degradation while also supporting the isolation of application components into easy to monitor, test, and debug components.

Liuba Shrira (session chair, Brandeis University) asked whether developers leveraging Fracture must manually define inter-module interaction in order for Fracture to compute an optimal partitioning. Pillai answered that no, a developer must only specify which modules must be kept isolated and provide the necessary annotations. Tomas Hruby (Vrije Universiteit) asked whether Fracture must maintain significant state in order to facilitate restarting FMPs, etc. Pillai answered that Fracture doesn't maintain significant internal state, but only requests state between module RPCs. This has some memory cost but does not impose significant computational overhead. Another attendee asked how correctness was affected when a program is split into multiple FMPs, e.g., when a single FMP crashes and is restarted. Pillai answered that Fracture requires the developer to build/divide modules in a manner that supports being safely restarted. Ken Birman (program chair, Cornell University) asked whether Fracture's concepts apply to languages like Java that have some existing tools for purposes similar to Fracture's. Pillai answered that there might still be value in porting the Fracture ideas to higher level languages, but that it is primarily designed for C, where no such tools exist.

Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications

Naila Farooqui, Georgia Institute of Technology; Christopher Rossbach, Yuan Yu, Microsoft Research; Karsten Schwan, Georgia Institute of Technology

Farooqui opened by introducing the audience to the basics of general purpose GPU computing, including presenting the GPU as a SIMD machine with a tiered memory hierarchy. GPU performance is often limited by two main factors: irregular control flow and non-sequential memory access patterns. Traditionally, optimizing GPU code to overcome these factors is difficult, requiring many manually applied architecture-specific tweaks. High level frameworks (i.e., Dandelion, Delite, etc.) exist to reduce the manual optimizing burden, but these frameworks still fail to capture the performance quirks of specific architectures, especially in terms of dynamically manifested performance bottlenecks. Farooqui et al. created Leo as an attempt to automate this performance-tuning process.

Leo automates performance tuning for GPUs by using profiling to optimize code for dynamic runtime behaviors. Farooqui presented a standard manual GPU performance optimization: Data Layout Transformation (DLT)—e.g., switching from row major to column major ordering to promote sequential memory access. To demonstrate the value Leo provides, Farooqui explained the challenge of applying the DLT optimization to an example program: SkyServer. In some cases the DLT optimization improves SkyServer performance, but in other cases performance is reduced. SkyServer requires dynamic optimization via a system like Leo to properly apply DLT when it helps while also forgoing it when it does not.

Leo employs dynamic instrumentation to drive runtime optimization. First, Leo generates a GPU kernel, then it analyzes,

instruments, and executes this kernel. Next, Leo extracts profiling results and identifies candidate data structures for optimization. Finally, Leo applies the identified optimizations. Leo iteratively repeats this process, regressing to the previous state if no benefits occur. Leo leverages GPU Lynx for instrumentation and Dandelion for GPU cross-compilation. Leo's performance approaches that of an "oracle"-derived solution (e.g., one that is hand-optimized for a known input) with gains from 9% to 53% over the unoptimized version. And since Leo's optimization is fully automated, these speedups are effectively "free."

Kishore Papineni (Google) asked whether optimization challenges in SkyServer were only related to cache misses. Farooqui answered that there were also likely other factors at play (e.g., control flow). Ken Birman asked whether compile-time analysis can provide better layout predictions than Leo's dynamic analysis. Farooqui answered that it may be possible, but there are many challenges since compile-time optimizations can't, for example, see cache impacts. Kornilios Kourtis (ETH Zürich) asked Farooqui to comment on data-flow models vs. other models (e.g., which are easier to optimize). Farooqui answered that you can profile and optimize using non-data-flow models, but that data-flow models make generating multiple code versions easier. John Reiser asked the bit width at which data coalescing happens and whether the number of workers affects performance. Farooqui tentatively answered that the width was 8 bytes, but needed to verify that. The optimal number of worker threads is hardware dependent. Liuba Shrira asked how Leo handles multiple independent optimizations while avoiding a combinatorial explosion of profiling results. Farooqui answered that applying multiple independent optimizations is complicated and that they are working on how best to handle such optimizations in Leo.

Proactive Energy-Aware Programming with PEEK

Timo Hönig, Heiko Janker, Christopher Eibel, Wolfgang Schröder-Preikschat, Friedrich-Alexander-Universität Erlangen-Nürnberg; Oliver Mihelic, Rüdiger Kapitza, Technische Universität Braunschweig

Hönig opened by highlighting the challenges developers face today trying to optimize their code for minimal power consumption. Modern hardware has many power-related levers (e.g., sleep states, toggling peripherals, processor throttling, etc.), but developers don't have good tools for creating code that optimally utilizes these levers. Instead, developers are limited to infinite iteration loops, requiring them to hand-tweak code for power consumption and then manually measure power consumption using either the limited built-in measurement systems or traditional tools like oscilloscopes and multimeters. The process of optimizing code for power consumption today involves (1) writing and compiling code, (2) running code with a well-defined set of inputs, (3) performing a manual energy usage analysis, and (4) optimizing code and repeating the cycle. This is a very time-consuming process for the developer.

Hönig et al. set out to improve this situation by integrating energy measurement and optimization suggestion support into

modern IDEs. They proposed a system for doing this called PEEK, a proactive energy-aware development kit. The PEEK framework is split into three parts: the front-end UI (e.g., Eclipse plugin, or CLI), the middleware that handles data storage and tracking, and the back-end energy analysis system and optimization hint engine. PEEK leverages the Git version control system to snapshot copies of code to be analyzed as well as various build-parameters. This allows developers to separate potential enhancements into separate Git branches and then use PEEK to analyze and compare the respective energy performance of each branch. Completed PEEK analysis results and potential energy optimization tips are also saved via Git. The front-end UI extracts these data and can automatically generate a source code patch that the developer may then choose to apply.

In addition to building the PEEK framework, the authors created a novel dedicated energy measurement device. Existing energy measurement devices tend to lack programmable control interfaces and the necessary measurement capabilities to produce accurate results. The authors' solution leverages a current-mirror to produce accurate energy measurements even at limited sampling rates. Their system uses an ARM Cortex-M4 MCU and contains a USB-based programmable control interface for fully automated energy measurement. Using the PEEK framework and their custom hardware, the authors were able to reduce developer energy optimization time by a factor of 8 while also reducing code energy consumption by about 25%.

Naila Farooqui (Georgia Institute of Technology) referenced some of Hönig's result graphs and asked why performance did not always map to power. Why do slower programs not always use less power? Hönig answered that they would have to look more closely at specific situations to know for sure. Kornilios Kourtis (ETH Zürich) commented that the optimality search base for power-use tweaks must be huge, including compiler flags, scheduling options, etc. Kourtis then asked whether there is additional support PEEK can provide to make it simpler for the developer to select the ideal set of optimizations. Hönig answered that future work will aim to tackle this problem and provide better developer hints. Jay York asked how PEEK synchronizes hardware measurements with code execution. Hönig answered that the system uses GPIO signals and relies on minimal overhead/cycles in the GPIO loop to keep measurements synchronized with code. Liuba Shrira asked whether PEEK can take advantage of common sections of the code across multiple snapshots to avoid extra analysis. Hönig answered that although their previous work explores that, PEEK's snapshot batching capabilities are primarily aimed at allowing developers to logically group potential changes, not at minimizing back-end analysis effort.

System Structuring

Summarized by Timo Hönig (thoenig@cs.fau.de)

From Feast to Famine: Managing Mobile Network Resources Across Environments and Preferences

Robert Kiefer, Erik Nordstrom, and Michael J. Freedman, Princeton University

Robert Kiefer presented Tango, a platform that manages network resource usage through a programmatic model.

Kiefer motivated his talk by demonstrating that network usage of today's mobile devices impact other system resources (i.e., data cap, battery, performance). Network resources should be allocated in different ways depending on dynamic conditions (e.g., if a user changes location) that cause different network technology (i.e., WiFi, 3G/LTE) to become available. Users' interests may also change over time (foreground vs. background applications), and network usage may depend on usage characteristics (e.g., interactive vs. streaming). Divergent goals between user and application trigger resource conflicts that have to be moderated by the user.

Using the example of a streaming music app, Kiefer further illustrated various conflicts between applications and between applications and users. He showed that one of today's mobile systems (Android) only exposes "all or nothing" controls that are unsuitable for efficiently managing resources. The available configuration options are application-specific, and the naming of the configuration options differs between applications. Users usually cannot control resource usage on their mobile phones as they want to.

In Tango, user and application configurations are encoded as policies. With these policies, user configurations have top priority, while application configurations have some flexibility. Kiefer further introduced the architecture of their framework (measure and control primitives, controller, user and application policies). Policies are actually programs that turn states into actions. A state of the system is used as input for a policy program to transform the system into a new state. Actions may impact network interfaces or network flows, where user policies may affect actions at interface level and flow level, but application policies may only affect actions at (their own) flow level.

With constraints and hints, Tango detects conflicts regarding resource usage at the user and application level. Hints for "future rounds" (e.g., higher network bandwidth) are matched with existing constraints.

The evaluation of Tango was demonstrated by a streaming music application used across a campus WiFi with unreliable data connection. They used this scenario to analyze when to switch between WiFi and the mobile network (3G/LTE). The evaluation scenario revealed certain switching problems and demonstrated which user policies to apply to optimize the 3G usage.

Gilles Muller (INRIA) asked how Kiefer would optimize applications with two competing policies. Kiefer replied that because of hierarchies (different classes of applications), his framework

does not have to know each single application. If an application's policy does not fit into a user policy, the user may either change the constraints or find a different application that fits the user's policy. Someone commented on the campus use case that showed areas of "good" and "bad" WiFi, and asked why a "good" WiFi actually is good and a "bad" WiFi actually is bad. Kiefer replied that "good" WiFi meant the user was close to the WiFi access point. Bad WiFi was usually when noise and transmission errors increased as the user moved out of reach. Ken Birman asked whether there was a policy that should say, "Don't use my cell connection if my buffer is under x percent." Kiefer explained that they have implemented this with a policy that restricts the amount of data that may be consumed during a specific amount of time. With this, Kiefer et al. were able to reduce cell usage by about 30%.

On Sockets and System Calls: Minimizing Context Switches for the Socket API

Tomas Hruby, Teodor Crivat, Herbert Bos, and Andrew S. Tanenbaum, Vrije Universiteit Amsterdam

Tomas Hruby motivated his talk by giving a quick overview on the problems of the POSIX Sockets API. While the POSIX Sockets API is well understood and portable across different operating systems, Hruby raised the concern that (1) each API call requires its own system call, (2) system calls are disruptive, (3) OS execution interleaves with applications, and (4) non-blocking system calls are expensive. Hruby further outlined how current solutions address issues of POSIX sockets. In contrast, Hruby et al. believe that the socket API itself is not broken. Instead, it is the underlying implementations that are broken. The system call itself is the key problem.

The approach presented by Hruby tackles the challenge of eliminating system calls from sockets while keeping the API intact. Their proposed solution is so-called exposed socket buffers, which allow applications to inspect socket buffer data directly without going through system calls. This way, most system calls could be handled in user space. An exposed socket buffer consists of a piece of shared memory between the OS and the application, two queues, and two memory areas for allocating data. With this data structure, an application can test in user space whether a socket is empty or full. Further, Hruby gave details on how signaling is implemented in their system. As a result of the system design, the network stack cannot easily poll without disturbing the applications. This is why the authors decided to move the network stack to a dedicated core.

The implementation is based on NewtOS, a multiserver system based on MINIX 3. Hruby gave numbers on the amount of messages required to complete a `recvfrom()` call that returns from EAGAIN on the modified NewtOS (137 messages) and compared it to Linux (478 messages). Hruby emphasized the improvement over the original NewtOS implementation (>19,000 messages).

For the evaluation, the authors used `lighttpd` (single process) serving static files cached in memory on a 12-core AMD 1.9 GHz system with a 10 Gigabit/sec network interface. During the pre-

sentation, Hruby showed numbers on the instruction cache miss rate of `lighttpd` where NewtOS (1.5 %) was performing better than Linux (8.5 %) and the unmodified NewtOS (4.5 %). Before concluding his talk, Hruby discussed the limitations of the presented approach (e.g., `fork()` is not supported) and presented related work.

Jie Liao (Rice University) asked how much more effort it takes to adopt applications to the programming model of the presented approach. Hruby replied that the application remained unchanged since the programming model is not changed at all. Jon A. Solworth (University of Illinois at Chicago) asked what would happen when you have large reads (i.e., megabyte reads) since caching and shared memory would be affected. Hruby replied that it really depends on the application and how the application processes the data. Kishore Pusukuri (Oracle Inc.) wanted to know how the system handles multithreaded applications such as `memcached`. Hruby referred to previous work and said that their system can run multithreaded applications just fine. Kishore inquired about the underutilization of available resources. Hruby answered that this is not an issue. Xinyang Ge (Pennsylvania State University) asked whether the presented design impacts the performance of applications which do not use the network. Hruby answered that this is not a problem because they use hyper-threading for the network stack and so the core is not lost for other operations.

Lightning in the Cloud: A Study of Transient Bottlenecks on n-Tier Web Application Performance

Qingyang Wang, Georgia Institute of Technology; Yasuhiko Kanemasa, Fujitsu Laboratories Ltd.; Jack Li, Chien-An Lai, and Chien-An Cho, Georgia Institute of Technology; Yuji Nomura, Fujitsu Laboratories Ltd.; Calton Pu, Georgia Institute of Technology

Qingyang Wang presented their study analyzing very short bottlenecks that are also bottlenecks with a very short life span (~ tens of milliseconds) and their impact on the overall system. According to Wang, very short bottlenecks are causing large response-time fluctuations for Web applications (10 milliseconds to 10 seconds).

Their study analyzed the reasons for very short bottlenecks in different system layers (system software, application, architecture) and investigated how such short bottlenecks can lead to delayed processing, dropped requests, and TCP retransmissions. Common for the analysis of all three system layers is a four-tier Web server (Apache/Tomcat/OJDBC/MySQL) running the RUBBoS benchmark, which emulates the workload of 24 users. Wang presented results that the Java Garbage Collector of the evaluation system caused very short bottlenecks in Tomcat. These short bottlenecks eventually lead to a push-back wave of queued clients of the Apache Web server. (Wang acknowledged that the Java Garbage Collector was fixed in JDK 1.7 and no longer suffers from the very short bottlenecks of Tomcat running with JDK 1.6.) Wang further demonstrated results of very short bottlenecks caused by bursty workloads in virtual machines, which eventually led to dropped requests and TCP retransmissions.

Someone asked whether there are generic solutions to avoid push-back waves caused by very short bottlenecks. Wang answered that there are two different solutions to address very short bottlenecks for the presented evaluation. First, very short bottlenecks in the first evaluation scenario can be avoided by upgrading the Java version. Second, very short bottlenecks in the second evaluation scenario can be avoided by migrating the affected virtual machine to a different machine. However, the authors are still working on a generic solution that addresses the problem by breaking up the push-back wave. Landon Cox (Duke University) asked how to generally diagnose the cause of very short bottlenecks. Wang replied that it is not easy to diagnose the cause and that it helps to apply fine-grained monitoring tools that collect as much data as possible. However, Wang admitted that there is no generic way to diagnose very short bottlenecks.

Security

Summarized by Ennan Zhai (ennan.zhai@yale.edu)

Custos: Increasing Security with Secret Storage as a Service

Andy Saylor and Dirk Grunwald, University of Colorado, Boulder

Andy presented Custos, an SSaaS prototype that can preserve encryption keys if customers store any encrypted data on the remote cloud side. Andy first described the Dropbox and Google Drive storage background: For current cloud providers, customers either trust the providers or store encrypted data on the cloud side but keep the key themselves or on other cloud storage providers. Both cases introduce privacy risks. Custos can provide the ability to securely store keys, and its goals are: (1) centralized secret storage, (2) flexible access control, and (3) auditing and revocation.

For each of the three goals, Andy showed a corresponding architecture graphic that was very illustrative. For centralized storage, Custos manages the key between different devices and sends the key to a centralized server. In addition, the authors leveraged a scalable SSL processor and multiple providers to maintain key shares. About this part, Andy said they applied an n -threshold cryptographic primitive to combine key shares, finally generating the desired key. For the flexible access control property, Custos allows the customers to write the access control specifications and ensure the security of the stored keys. Using this mechanism, customers can control the access time of the keys and have time-limited access capability. For the final property of Custos, i.e., auditing and revocation, the system can audit logs and keep track of key access, thus offering auditing and revocation capability.

Ennan Zhai (Yale) asked where the maintainers hold key shares in practice, and who produces the access control specification. Andy said in practice there are many individual companies that offer services maintaining such shares, so Custos can distribute the shares to them. For the second question, Andy thought the customers can use existing tools or experts to achieve their goals; in practice it is not so hard to do. Someone noted that since

Dropbox can share data with some mobile devices, it is harder for random third-party mobile applications to handle that. Andy said in principle it is not a concern for Custos, since the Custos system can flexibly handle such things.

Managing NymBoxes for Identity and Tracking Protection

David Wolinsky, Daniel Jackowitz, and Bryan Ford, Yale University

David Wolinsky began by noting that current anonymity tools (Tor) still cannot provide perfect privacy protection, and there have been many examples of an adversary focusing on breaking the user environment and not the tool. From this observation, the authors proposed Nymix, an operating system that can offer isolation for each browser task or session. Each browser or session is running in a virtual machine called Nym.

David used three interesting examples to describe the three target problems, including application-level attacks, correlation attacks, and confiscation attacks, which can expose users' privacy even if they use Tor. David then presented Nymix architecture design: In general, each Nym has two components: AnonVM and CommVM. AnonVM is mainly responsible for running the actual applications the user wants to run, while CommVM communicates with the outside environment. Since Nymix offers virtual machine isolations, the user environment cannot be compromised by the three types of attacks described above.

In the evaluation, David showed how the prototype can be set up as well as covering CPU overhead, memory usage, and networking overhead. Finally, David talked a lot about future improvements on Nymix, including fingerprintable CPU, VMM timing channels, accessing local hardware, and storing data retrieved from the Internet.

Someone asked about a fingerprintable CPU and whether the authors had tried any experiments on this level. David thought in principle the results were not hard to anticipate, but it was still an interesting direction. The session chair asked about malicious virtual machines that can communicate with each other, compromising privacy defenses. David said that basically the paper does not need to consider such a case, the assumption being that the virtual machine manager is trusted.

10th Workshop on Hot Topics in System Dependability

October 5, 2014, Broomfield, CO

Summarized by Martin Küttler, Florian Pester, and Tobias Stumpf

Paper Session 1

Summarized by Florian Pester (florian.pesther@tu-dresden.de) and Tobias Stumpf (tobias.stumpf@tu-dresden.de)

Compute Globally, Act Locally: Protecting Federated Systems from Systemic Threats

Arjun Narayan, Antonis Papadimitriou, and Andreas Haeberlen, University of Pennsylvania

Cascading failures can bring down whole systems, and in a world of critical systems, this should be avoided. However, privacy concerns often prevent the global view necessary to detect impending catastrophic cascades. The authors studied this problem using the example of the financial crisis of 2008.

Banks usually have financial risks that are greater than their capital, therefore the surplus risk—the difference between their risk and their capital—is offloaded to other banks. This creates a network of dependencies between banks that results in a very large dependency graph. However, each bank only has its local view of the situation.

System-wide stress tests could be a solution to the problem; unfortunately, a lot of the input needed for such a test produces privacy issues for the banks. Therefore, economists do not know how to compute a system-wide stress test—although they do know what to compute. A regulator is not an option, because a regulating entity would require too much access. Secure multi-party computation, on the other hand, does not scale well enough.

A workable solution must deal with two main challenges: privacy and scalability. In order to provide scalability, a graph-based computation is used instead of matrix multiplication. Each bank is assigned a node and performs what the authors call limited multi-party computation.

In order to solve the privacy problem, the authors use differential privacy, which provides strong, provable privacy guarantees. This works by adding a small amount of imprecision to the output of the computations, which can be tolerated because the aim of the computation is detection of early warnings of large problems. Limited multi-party computation is essentially multiple multi-party computations with k parties. Each party gets another party's output as input for their own computation; in this way, no party has access to the other party's secrets.

Implementation was left as future work.

Someone commented that this work could also be applied for the power grid and other dependable systems. Additionally, the question was raised whether it is a realistic assumption that the banks will follow the protocol. The answer was that the regulators get localized data and can enforce the protocol.

Running ZooKeeper Coordination Services in Untrusted Clouds

Stefan Brenner, Colin Wulf, and Rüdiger Kapitza, Technische Universität Braunschweig

Privacy issues and insufficient trust in cloud providers slow down the adoption of cloud technologies. However, computation of data is tricky if that data is encrypted. Trusted Execution Environments (TEE) can help to mitigate the problem.

Apache ZooKeeper provides coordination for cloud applications. The authors added an encryption layer to ZooKeeper in order to gain trusted coordination.

This encryption layer is provided in the form of the ZooKeeper privacy proxy, which runs inside a TEE. Communication from the client to the ZooKeeper privacy proxy is encrypted by SSL, while communication between ZooKeeper and the ZooKeeper privacy proxy is encrypted using a shared key between these two. Name-clashing problems are solved by a dictionary node.

Someone asked the presenter to clarify whose signing key needs to be trusted ultimately. The answer was the hardware signing key. A second questioner wanted to know more about the applications that can be used with the system. The presenter answered that the system needs specific applications. The final questioner wondered why the system was implemented as a proxy. The reason is so that this solution is completely transparent to the server and the client.

Who Writes What Checkers?—Learning from Bug Repositories

Takeshi Yoshimura and Kenji Kono, Keio University

Takeshi and Kenji presented a tool that learns from bug repositories to help developers eliminate bugs. The tool uses static code analysis to find typical bugs—e.g., not freed memory or infinite polling—and is based on the Clang analyzer. During analysis Takeshi figured out that bugs coming from humans are mainly domain specific. Their tool uses machine-learning techniques to extract bug patterns from documentations written in English and the corresponding patches. The tool works in two steps: First, it uses natural-language processing to extract keywords from the documentation. Second, their tool extracts bug patterns based on topic. For the paper, Takeshi analyzed 370,000 patch documentations and sorted them into 66 groups called “clusters.” They also found subclasses by using keywords; for instance, around 300 patches contain the keyword “free.” They use the known free-semantic to check patches and see whether a free statement is missing. The presented tool is useful for detecting typical bugs. In his conclusion, Takeshi mentioned that finding unknown bugs is more complicated but possible.

Someone asked about the documentation used. Takeshi clarified that they used only the commit messages and not information from the bug repository. Further questions were related to language processing. Takeshi explained that they used techniques to reduce language noise from developers' commit messages.

The last questioner wanted to know which kind of bug patterns they could find and which not. The presenters clarified that their tool is limited to known bugs which are already documented.

Leveraging Trusted Computing and Model Checking to Build Dependable Virtual Machines

Nuno Santos, INESC-ID and Instituto Superior Técnico, Universidade de Lisboa; Nuno P. Lopes, INESC-ID and Instituto Superior Técnico, Universidade de Lisboa and Microsoft Research

Nuno Santos and Nuno Lopes developed an approach based on trusted booting and model checking to ensure that a virtual machine (VM) in the cloud runs the software that a customer expects. The basic approach to launch a VM is quite easy. Someone creates a VM image and sends it into the cloud, and someone (maybe the image creator) launches the VM. Previous studies showed that this approach includes risks for the creator as well as for the user. For instance, a misconfigured VM image can contain creator-specific data (e.g., passwords, IDs) or include obsolete or unlicensed software. To overcome these problems, Nuno proposed model checking to ensure that a VM image matches the specifications before it is launched. The specification includes all necessary information (e.g., configurations, applications) to ensure a specification behavior of the VM. After a VM is checked, a trusted computing environment is used to ensure that the checked image is launched.

Someone wanted to know how to identify passwords. Nuno answered that it is done by annotations. A second questioner asked who he has to trust (machine, cloud provider) to launch a VM. The software integration can be done outside the cloud, and therefore it is not necessary to trust any cloud provider. Another question related to software specification in general. Nuno answered that their work does not focus on software verification but on checking the properties of a given instance. Finally, to the question of how much work it takes to write a specification, Nuno said the workload depends on the level of fine tuning. It is simple to write a running specification, but the tuning part can be quite expensive.

Paper Session 2

Summarized by Martin Küttler (martin.kuettler@os.inf.tu-dresden.de)

Erasure Code with Shingled Local Parity Groups for Efficient Recovery from Multiple Disk Failures

Takeshi Miyamae, Takanori Nakao, and Kensuke Shiozawa, Fujitsu Laboratories Ltd.

Takeshi Miyamae began by noting the need for erasure codes with high durability and an efficient recovery. He presented a Shingled Erasure Code (SHEC) that targets fast recovery and correctness in the presence of multiple disk failures.

He discussed the three-way tradeoff between space efficiency, durability, and recovery efficiency. SHEC targets recovery efficiency foremost by minimizing the read sizes necessary for recovery. Compared to Reed Solomon, MS-LRC, and Xorbas, SHEC has substantially better theoretical recovery speed for multiple disk failures.

Next, Takeshi showed that SHEC is comparable to MS-LRC in terms of durability for double disk failures, but is more customizable and offers more fine-grained tradeoffs in efficiency. He then briefly explained the implementation of SHEC, which is a plugin for the free storage platform Ceph.

Takeshi presented an experiment comparing SHEC to Reed Solomon for double disk failures. He found that SHEC's recovery was 18.6% faster and read 26% less data from disk. Then he showed that there was 35% more room for recovery time improvement because the disks were the bottleneck only 65% of the time.

Someone asked why the disks were not always the bottleneck during the experiment. Takeshi explained that other resources can be the bottleneck too, but in their experiment only the disk was a bottleneck. He was not sure what acted as bottleneck during the remaining 35% of the experiment, but it was not the CPU or the network.

The second questioner wondered how common it is to have multiple disk failures. Takeshi answered that the probability for that is higher in practice than in theory.

Providing High Availability in Cloud Storage by Decreasing Virtual Machine Reboot Time

Shehbaz Jaffer, Mangesh Chitnis, and Ameya Usgaonkar, NetApp Inc.

Shehbaz Jaffer presented work on providing high availability in cloud storage. Virtual storage architectures (VSA)—storage servers that run as virtual machines on actual hardware servers—are considerably cheaper and more flexible than hardware servers. But they suffer from lower availability, because in case of failures they need to be rebooted, which introduces long wait times. The typical solution for hardware servers is to deploy high availability (HA) pairs, i.e., two servers that both serve requests, so that one can keep working when the other reboots.

To achieve the goal of this work, reducing the VSA reboot time in order to increase availability, Shehbaz presented multiple steps. The first was to cache VM metadata in the host RAM and provide access to that data on reboot. This improved boot time by 5%, which was less than was expected.

Next the authors tried to improve the SEDA architecture to improve performance. In particular, synchronously returning cached file-system information, instead of performing an asynchronous call, reduced boot time by 15%.

Finally, they improved block-fetching time during a reboot. Turning read-ahead during reboot decreased the boot time by another 18%.

The time breakdown showed that consistent checkpointing takes a lot of time. The authors left replacing consistent checkpointing with a faster technique as an open problem, which they want to look at in follow-up work.

Somebody asked how often VSA failures, and subsequent reboots, happen in practice. Shehbaz answered that about 85–90% of the failures are software failures, such as OS bugs or mishandled client requests. But he had no absolute numbers. Another questioner asked why not use multiple replicated VMs, which can take over when one fails. Shehbaz answered that in deploying a VSA at some datacenter, there is no way to ensure that the VMs are going to be near each other, and maintaining consistency is much harder when they aren't.

Understanding Reliability Implication of Hardware Error in Virtualization Infrastructure

Xin Xu and H. Howie Huang, George Washington University

As motivation for their work, Xin Xu pointed out that hardware failures are a common problem in the cloud: If a server has one failure in three years, a datacenter with 10,000 servers has an average of nine failures per day. Still, the datacenter provider needs to handle these incidents to provide a reliable service.

In his work, Xin focused on soft errors such as transient bit flips, which are not reproducible and are hard to detect. With smaller process technologies, the probability for transient failures is expected to increase greatly, which is a major reliability concern in datacenters.

He also pointed out that hypervisors are a single point of failure and do not isolate hardware errors. They studied this problem by doing fault injections into virtualized systems. For these experiments, the authors analyzed the hypervisor to find the most used functions and found that by injecting errors in them, they could find more crashes than by doing random injections. Using this analysis, they also found more errors that propagated into the VMs.

In addition, they categorized the injected faults based on the crash latency, which is the number of instructions between fault injection and detection of the crash. Most crashes have a short latency of fewer than 100 instructions, but some are significantly longer and are thus likely to propagate into a VM. To study this they also analyzed failure location. In many cases, crashes happen in the C-function where a fault was injected. A small percentage (up to 5%) leave the hypervisor, meaning there is no fault-isolation.

Comparing his work to previous approaches, Xin highlighted two new contributions: a simulation-based fault injection framework, and analysis of error propagation.

Somebody asked whether they considered bit flips in opcodes as well as in program data. Xin answered that they only injected bit flips into registers, because injecting errors into memory is more difficult to track. He was then asked whether he considered storage more reliable. He responded that he did not, but that memory is protected by ECC, and that errors in the CPU are more difficult to detect. Somebody asked how Xin would go about making the hypervisor more robust, and what implications their framework had on performance. For the first question, Xin referred the questioner to another paper on ICPP about detect-

ing failures early, before they propagate to dom0. Regarding the performance, he said that the overhead was generally less than 1% because they used a lot of hardware support such as perf counters.

Towards General-Purpose Resource Management in Shared Cloud Services

Jonathan Mace, Brown University; Peter Bodik, Microsoft Research; Rodrigo Fonseca, Brown University; Madanlal Musuvathi, Microsoft Research

Jonathan Mace presented the motivation of the work, describing how performance problems can arise in shared-tenant cloud services. Requests can drain resources needed by others. The system typically does not know who generates requests, nor which requests might interfere, and therefore it can give no performance guarantees to users. Ideally, one would like quality-of-service guarantees similar to those provided by hypervisors.

To address these issues, Jonathan proposed design principles for resource policies in shared-tenant systems. He then presented Retro, a prototype framework for resource instrumentation and tracking designed according to these principles.

He started by discussing interferences of various file-system operations on an HDFS installation. He presented measured throughput rates of random 8k writes with different background loads. Even tasks like listing or creating directories interfered significantly with the writer task. From that the authors inferred the first principle: Consider all request types and all resources in the system.

Jonathan motivated the second principle by discussing latency requirements of specific tasks. He provided an example where a high priority tenant required a low latency, and two tenants with lower priority were running. Throttling one of them could significantly decrease the latency of the high priority tenant, while throttling the other did not change the latency at all. Therefore only the correct tenant should be throttled, which led to the second principle: Distinguish between tenants.

Jonathan discussed long requests and the problems they pose for giving guarantees to other tenants, which generated the third principle: Schedule early and often. This avoids having long-running requests taking up resources that should be available for high priority tenants.

Finally, Jonathan presented Retro, a framework for shared-tenant systems that can monitor resource accesses by each request. The framework tracks which tenant issued each request and resource access, aggregates statistics (thus providing a global view of all tenants), and allows a controller to make smart scheduling decisions. He revisited the example of the high priority tenant interfering with one of two low priority tenants. Retro figures out which low priority tenant it has to throttle to improve the latency of the higher priority tenant. Jonathan also presented measurements showing that Retro introduced less than 2% overhead.

Somebody asked about the level of isolation that Retro provides and whether crashes were considered. Jonathan answered that Retro can provide performance isolation but no strong guarantees, as only averages can be guaranteed. He also said that they did not consider crashes.

Scalable BFT for Multi-Cores: Actor-Based Decomposition and Consensus-Oriented Parallelization

Johannes Behl, Technische Universität Braunschweig; Tobias Distler, Friedrich-Alexander-Universität Erlangen-Nürnberg; Rüdiger Kapitza, Technische Universität Braunschweig

Johannes Behl presented a scalable Byzantine fault tolerance implementation for multicore systems. He motivated this work by saying that many systems targeting dependability only consider crashes, which is not enough. To be able to handle more subtle faults, Byzantine fault tolerance (BFT) is needed. But it is not common, because current BFT systems do not scale well with multiple cores, which is required to run efficiently on modern hardware.

Johannes talked about state-of-the-art BFT systems and their parallelization. The BFT protocol consists of an agreement stage and an execution stage, which need to be parallelized. The parallelization of the second stage depends on the service characteristics. For the agreement stage, Johannes briefly presented the current method of parallelization. The agreement is split into multiple tasks, each running in a different thread. Thus, finding a consensus involves all threads, which has a number of disadvantages. The number of threads is determined by the BFT-implementation and is therefore not well aligned with the number of cores. In addition the slowest task inside the agreement phase dictates the performance. And the need for frequent synchronization between threads further increases the time to find consensus.

Johannes presented consensus-oriented parallelization, which involves vertical parallelization, i.e., having each thread serve a complete agreement request. That way throughput rates can scale with the number of CPU cores, and synchronization is much less frequent.

In the evaluation, Jonathan compared his implementation to a state-of-the-art implementation of BFT. His implementation scales much better, and it is already about three times faster for a single core, because it does not have any synchronization overhead there.

Someone asked how the logging inside the agreement stage was parallelized. Johannes answered that they did not log to disk but only to memory. For disk logging he proposed using multiple disks or SSDs. The next question was how dependencies across requests were handled. Johannes answered that they made no assumptions about dependencies. Somebody asked whether the evaluation was done in the presence of faults so that non-trivial agreements had to be found. Jonathan answered that there were no faults or disagreements in the evaluation. The last question was whether batching was used in the evaluation. Jonathan said no and noted that batching can improve performance.

REAL SOLUTIONS FOR REAL NETWORKS

FREE DVD

zentyal Small Business Server
64-bit Community Edition 3.5

10 Tips for Better SSL

ADMIN
Network & Security

FREE DVD!

ADMIN

Network & Security

DEC 2014/JAN 2015

10 Tips for Better SSL

Best practices for managing website risk

PORT KNOCKING
Don't miss this cool trick for securing remote access

Ceph or GlusterFS?
Choosing a shared storage alternative

Free AD Tools
Manage your Active Directory on a budget

Top Tools

FIDO Alliance
Exploring the future of authentication

• Unix • Solaris

Log Parser Studio
Reading logfiles on Windows networks

zine.com

ADMIN
DEC/JAN 2015
US\$ 15.99
CAN\$ 17.99
01



0 74470 86640 4



Each issue delivers technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- system management
- troubleshooting
- performance tuning
- virtualization
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

**FREE
CD or DVD
in Every Issue!**

6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com



USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER

Send Address Changes to *login*:
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE

PAID

AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

Interested in Site Reliability Engineering?

SREcon is back!

SREcon15

MARCH 16–17, 2015
SANTA CLARA, CALIFORNIA, USA
www.usenix.org/srecon15

SREcon15 EUROPE

MAY 14–15, 2015
DUBLIN, IRELAND
www.usenix.org/srecon15europe

Following 2014's inaugural sold-out conference, SREcon has expanded to two venues for 2015.

If you already work in an SRE environment—or want to learn how it's being used by many of the largest companies today—take advantage of this rare opportunity to meet with other engineers and discuss tricks of the trade.

Register today at www.usenix.org

